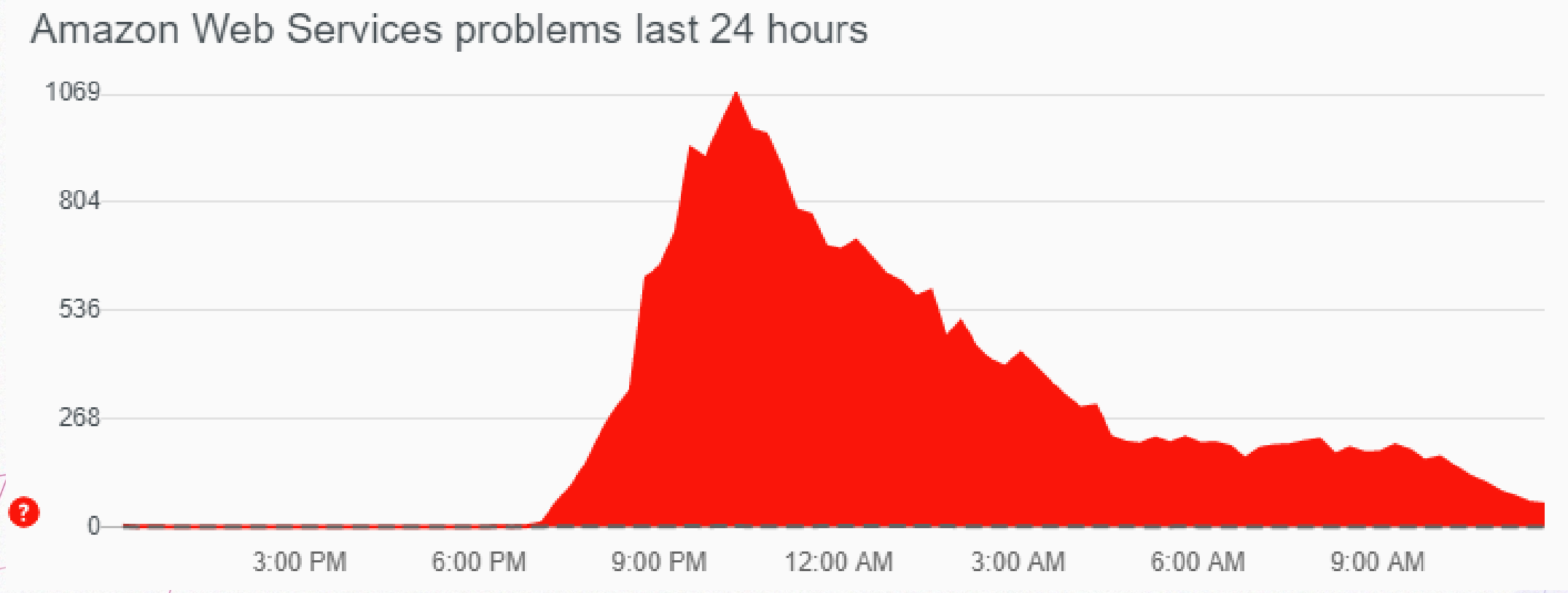
Arquitectura en la Nube

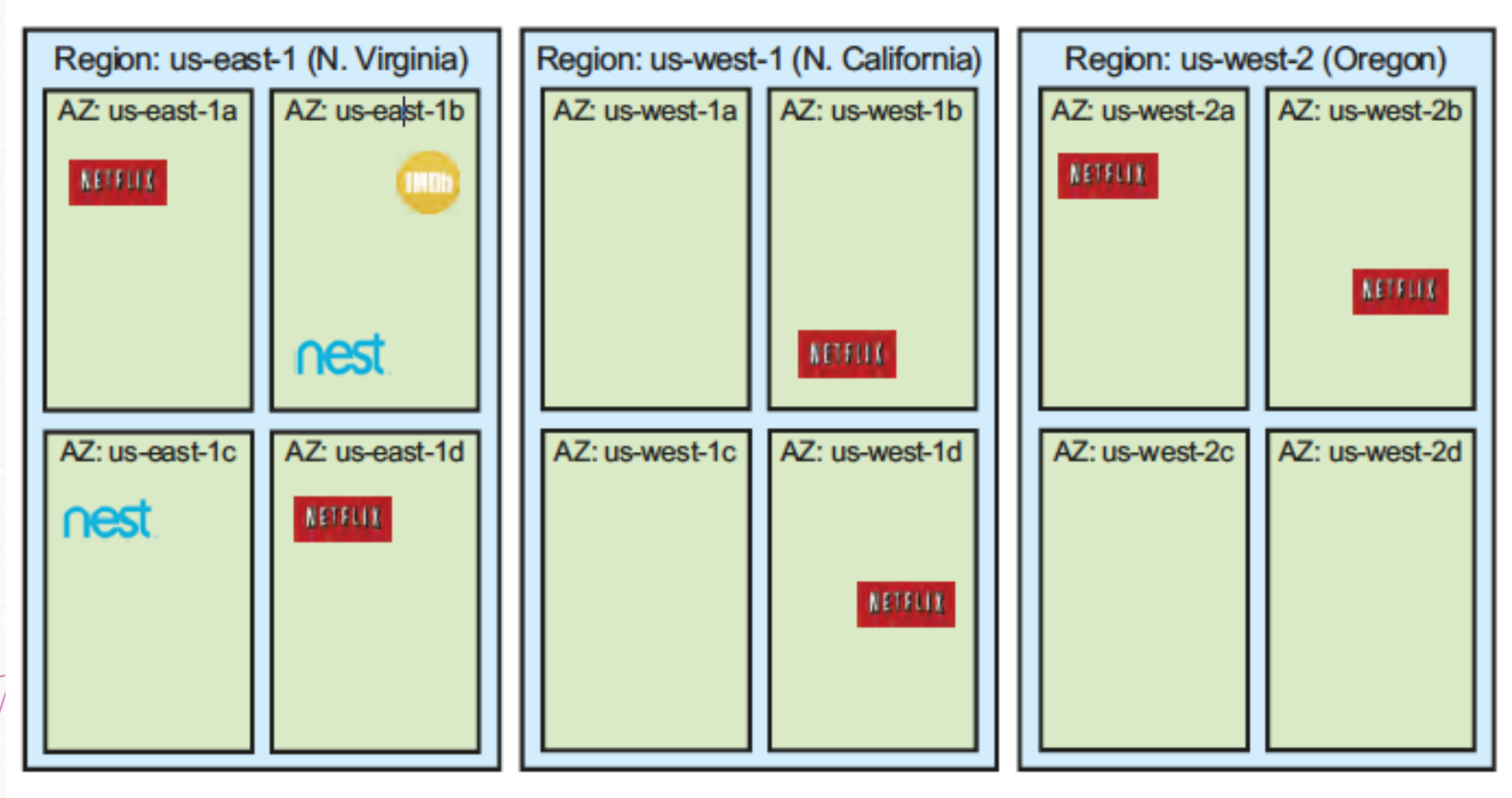
“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

— Grady Booch

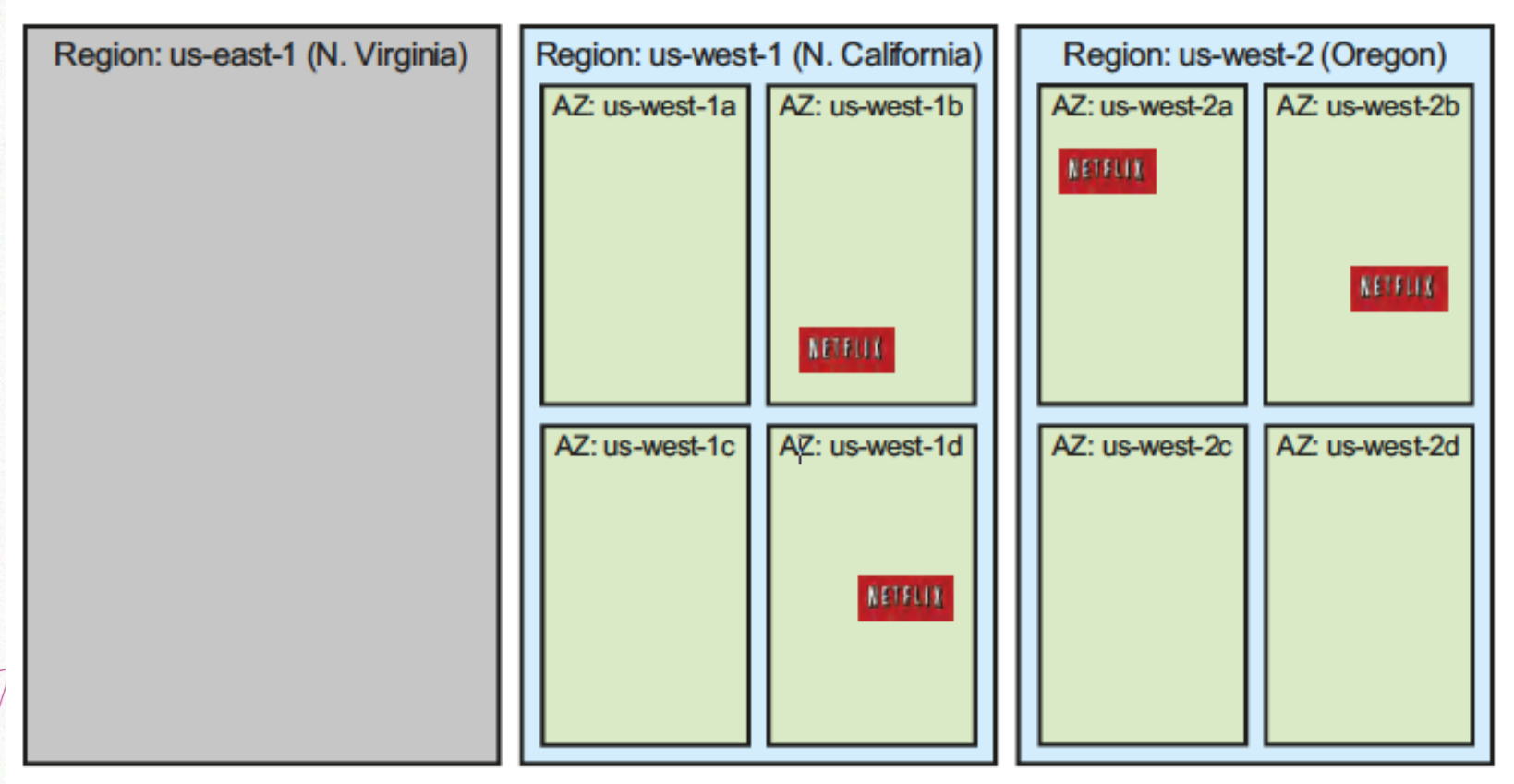
Septiembre 2015



Septiembre 2015



Septiembre 2015



Diseño Nativo para la Nube vs Despliegue en Nube

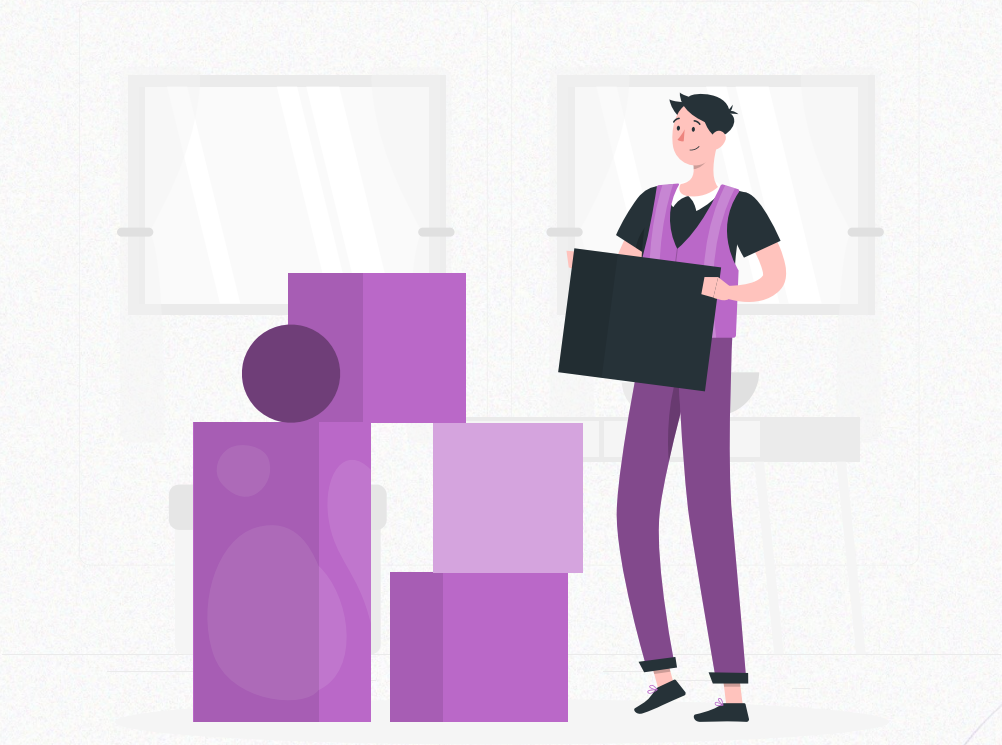
- Zero Downtime
- Ciclos de desarrollo cortos
- Ubicuidad
- Internet de las cosas
- Data Driven Development

Conceptos

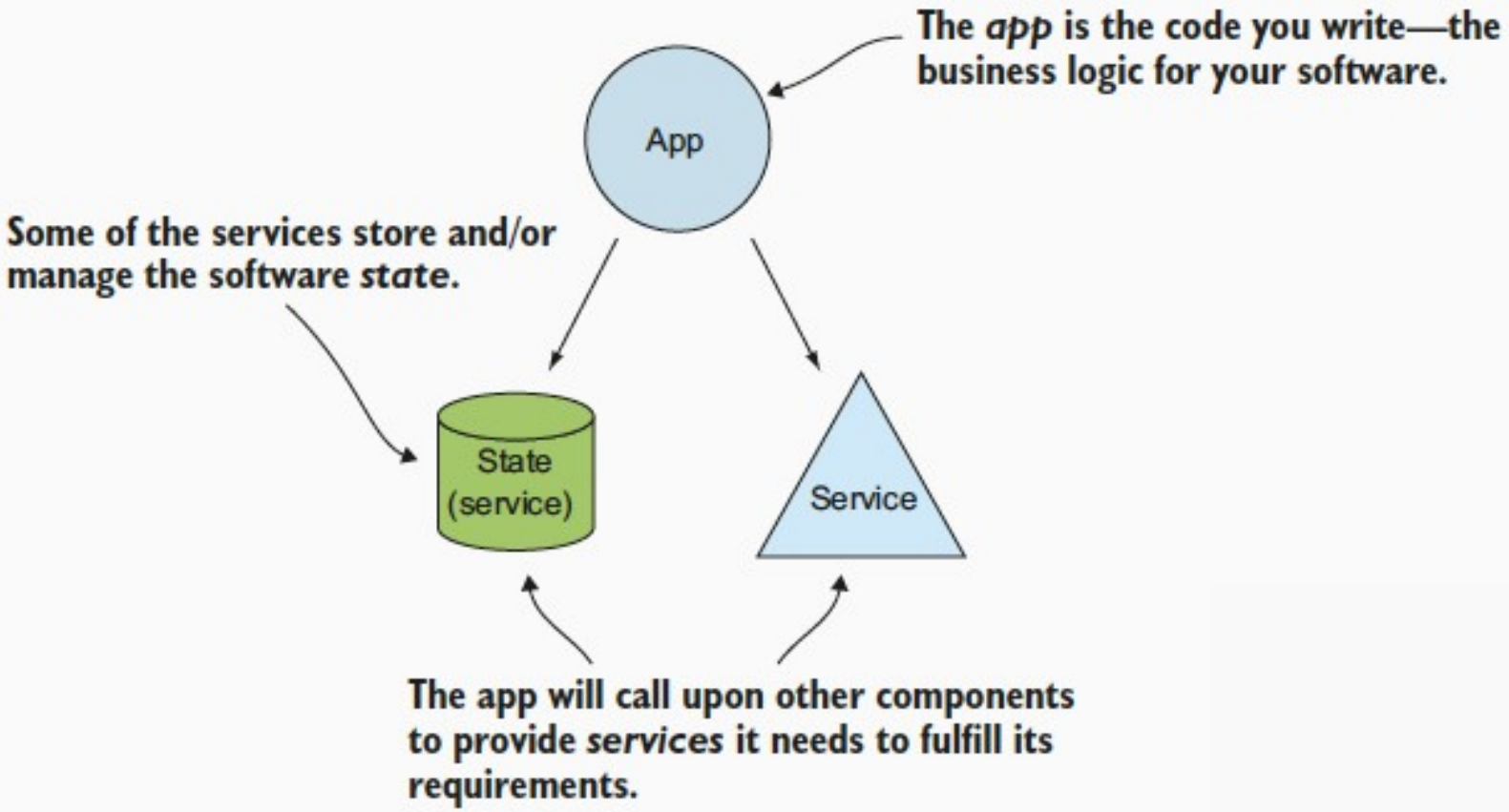
App Cloud Native: Se refiere al conjunto de la lógica de negocio y las funcionalidades que componen el software.

Cloud-native data: Es el modelo de datos y la forma en que se persiste la información en una arquitectura nativa de la nube.

Cloud-native interactions: Son las comunicaciones o intercambios que ocurren entre los distintos servicios (las *cloud-native apps*) y los sistemas de datos (las *cloud-native data*) en un entorno altamente distribuido.



Conceptos


The *app* is the code you write—the business logic for your software.
App
Some of the services store and/or manage the software *state*.
State (service)
Service
The app will call upon other components to provide *services* it needs to fulfill its requirements.

Cloud-native app: el lugar donde vive la lógica de negocio

- **No hay una “app” aislada** que cumpla todas las funciones de un sistema complejo. Por el contrario, hay muchos servicios trabajando en conjunto para ofrecer una solución digital completa.
- Cada servicio debe **aplicarse patrones nativos de la nube** para aprovechar elasticidad, escalabilidad y orquestación.
- El diseño de cada servicio o componente debe permitir la **interoperabilidad** con otros, facilitando la adopción de prácticas como despliegues ágiles, resiliencia y upgrades sin afectar el resto de la arquitectura.

Cada servicio debe tener un **propósito claro** y se diseña bajo principios como el de *12-Factor App*

12 – Factor App

I. Código base (Codebase)

Un código base sobre el que hacer el control de versiones y multiples despliegues

II. Dependencias

Declarar y aislar explícitamente las dependencias

III. Configuraciones

Guardar la configuración en el entorno

IV. Backing services

Tratar a los “backing services” como recursos conectables

V. Construir, desplegar, ejecutar

Separar completamente la etapa de construcción de la etapa de ejecución

VI. Procesos

Ejecutar la aplicación como uno o más procesos sin estado

VII. Asignación de puertos

Publicar servicios mediante asignación de puertos

VIII. Concurrencia

Escalar mediante el modelo de procesos

IX. Desechabilidad

Hacer el sistema más robusto intentando conseguir inicios rápidos y finalizaciones seguras

X. Paridad en desarrollo y producción

Mantener desarrollo, preproducción y producción tan parecidos como sea posible

XI. Historiales

Tratar los historiales como una transmisión de eventos

XII. Administración de procesos

Ejecutar las tareas de gestión/administración como procesos que solo se ejecutan una vez

Cloud-native data: repensando el estado en la nube

- Históricamente, se usaba un modelo de base de datos monolítica para todo el sistema. Es decir, un solo repositorio que almacenaba perfiles de usuario, pedidos, inventario, transacciones, etc.
- En un modelo nativo de la nube, cada servicio/microservicio puede tener su propia fuente de datos, de modo que la base de datos también se “rompe” en partes más pequeñas y autónomas.

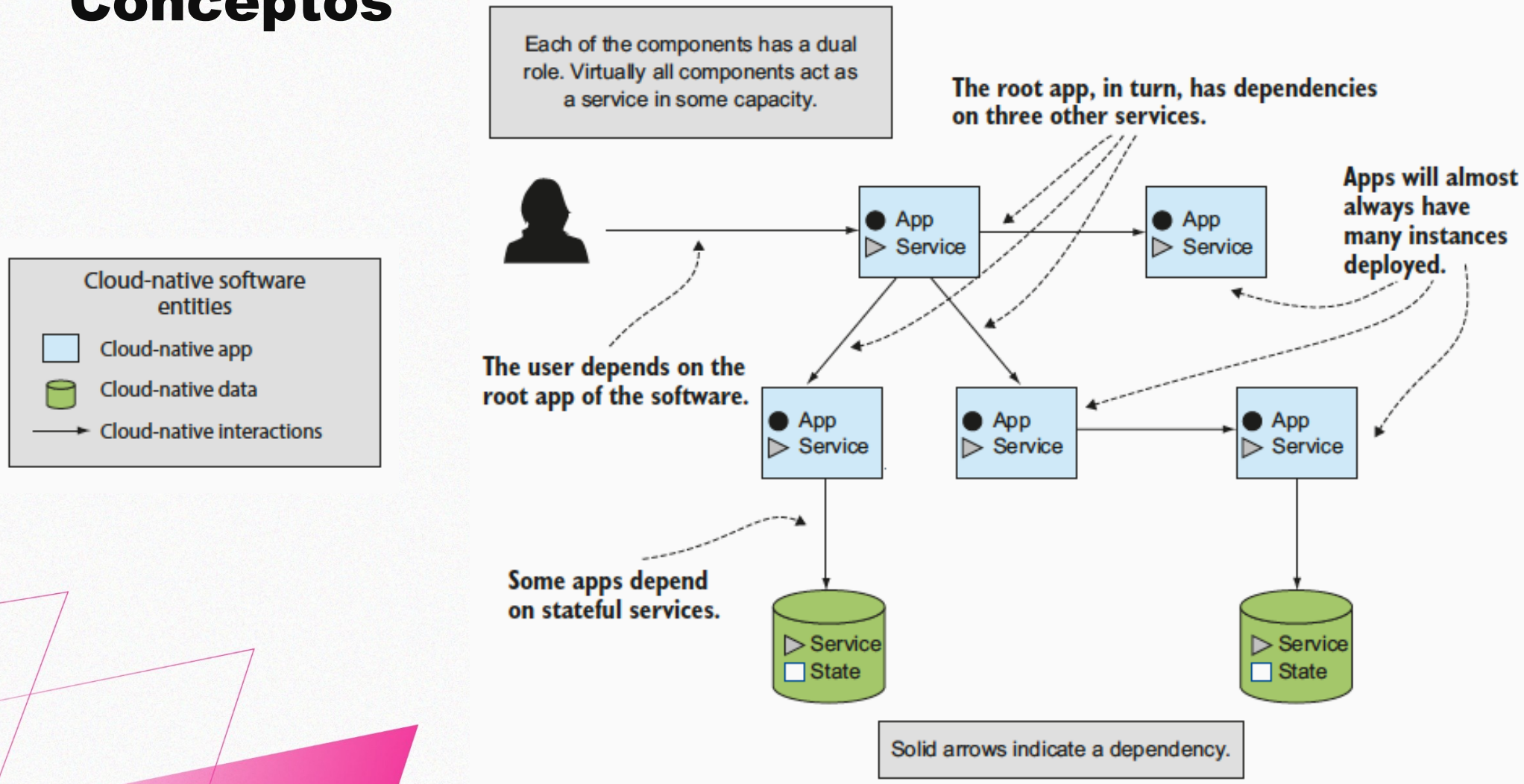
Cloud-native interactions: comunicación en un entorno distribuido

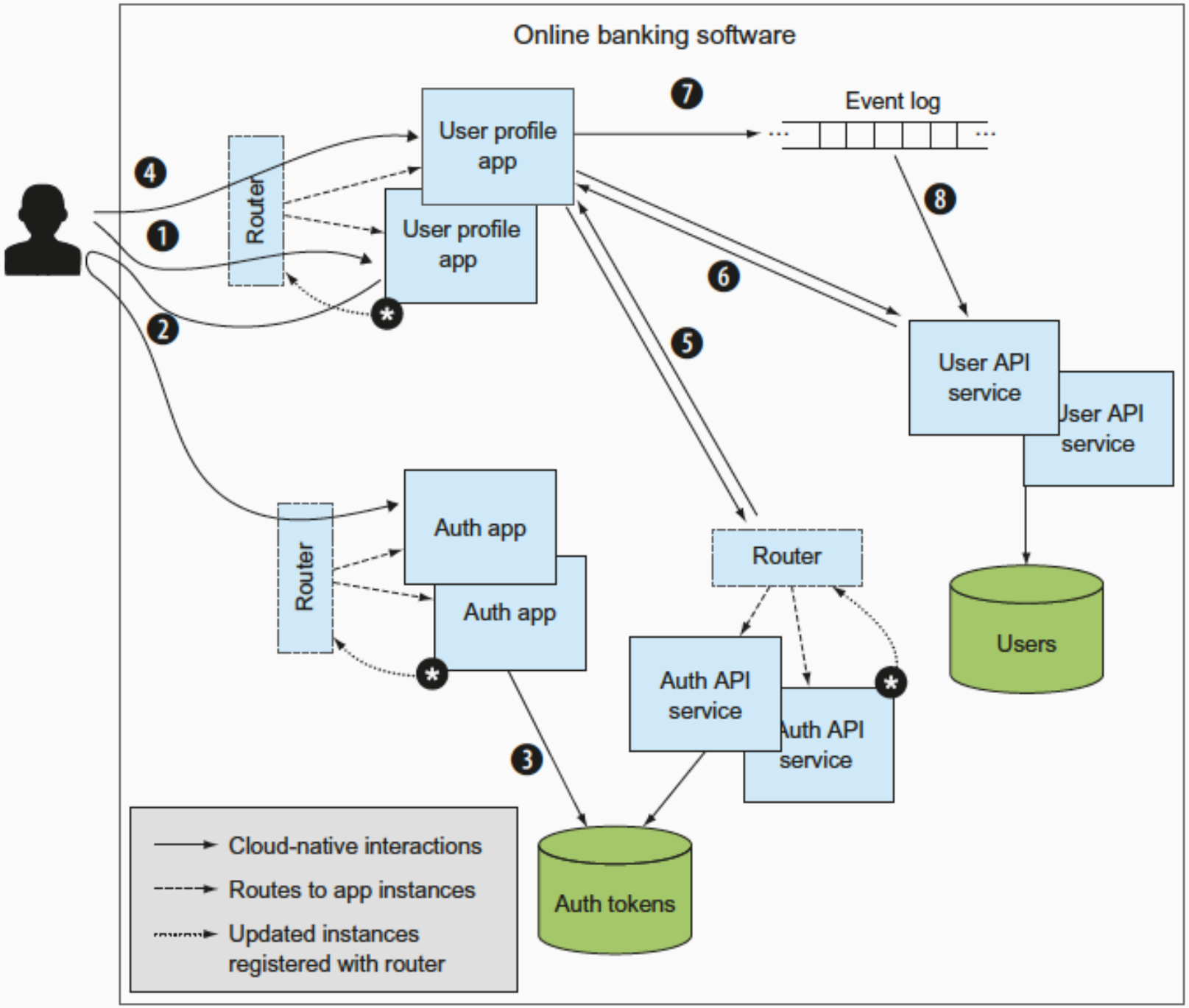
Dada la **naturaleza distribuida** de estos sistemas, las interacciones también han tenido que evolucionar:

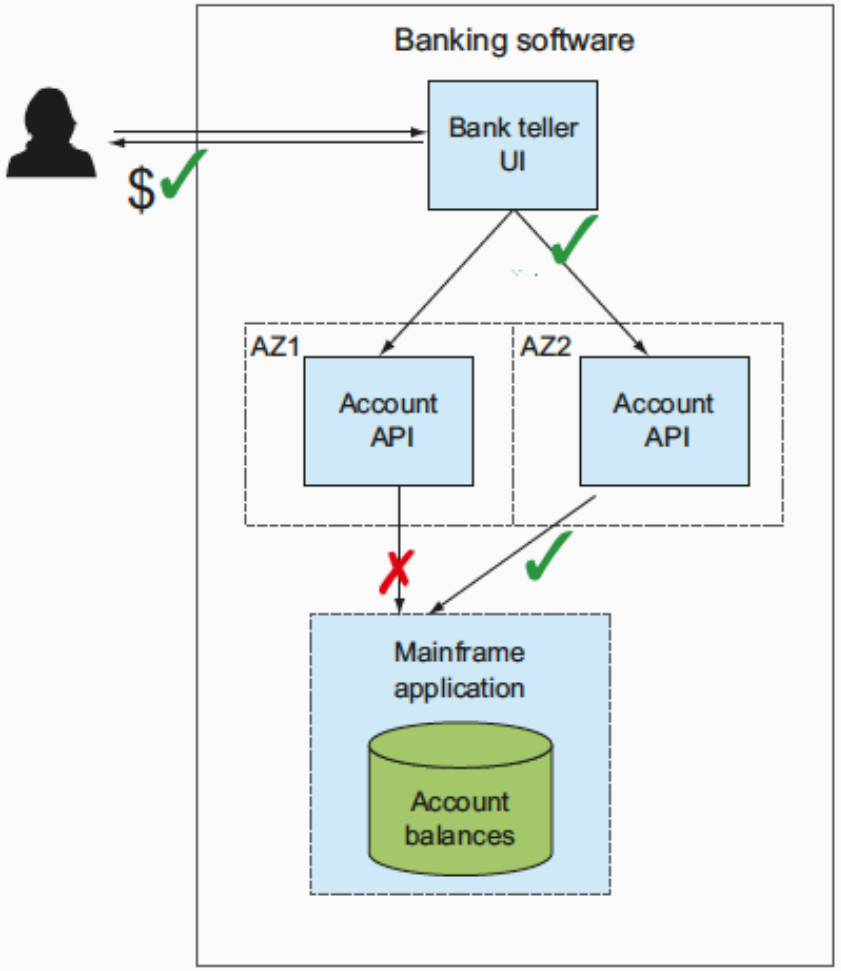
Patrones de comunicación

- **Síncrona**: Llamadas HTTP/REST o gRPC entre servicios, con enrutamiento y balanceadores de carga.
- **Asíncrona**: Uso de colas, buses de mensajería o flujos de eventos (Kafka, RabbitMQ) para desacoplar productores y consumidores.
- **Event-Driven**: Cada vez cobra más importancia la arquitectura orientada a eventos, donde un servicio emite un suceso y otros reaccionan.

Conceptos


Each of the components has a dual role. Virtually all components act as a service in some capacity.
The root app, in turn, has dependencies on three other services.
Apps will almost always have many instances deployed.
The user depends on the root app of the software.
Some apps depend on stateful services.
App
Service
Service
State
Solid arrows indicate a dependency.
Cloud-native software entities
Cloud-native app
Cloud-native data
Cloud-native interactions


Online banking software
User profile app
User profile app
Router
Event log
User API service
User API service
Router
Auth app
Auth app
Router
Auth API service
Auth API service
Users
Auth tokens
Cloud-native interactions
Routes to app instances
Updated instances registered with router


Banking software
Bank teller UI
$
AZ1
AZ2
Account API
Account API
Mainframe application
Account balances

¿Han tenido alguna experiencia con arquitecturas que fueron difíciles de escalar o mover a la nube?

¿Por qué creen que pasó?

Cloud Native no es solo infraestructura, es un *mindset* y un conjunto de prácticas para que las aplicaciones aprovechen las ventajas de la nube: escalabilidad, resiliencia, y entrega continua.

Foundation Patterns

Cornelia Davis denomina “Foundation Patterns” a aquellos patrones que resultan **cruciales** y **transversales** a toda aplicación *cloud native*. Son la piedra angular sobre la cual se construye el resto de la arquitectura.

Si bien la lista concreta de patrones puede variar según la fuente, en este capítulo la autora se enfoca en varios puntos clave:

Stateless Services

External Configuration

Declarative Deployment

API-Driven Contracts

¿Por qué creen que mantener estado en un servicio puede ser un problema en la nube?

Stateless Services

Un servicio “sin estado” no conserva información en memoria o en disco local que sea necesaria para el funcionamiento de la aplicación a largo plazo.

El objetivo es que cada instancia del servicio pueda escalarse, reiniciarse o reemplazarse sin perder datos, pues la información crítica está en bases de datos externas o servicios de almacenamiento.

Esto facilita la **orquestación** en entornos como Kubernetes y habilita el autoscaling dinámico.

Ejemplo

Una API RESTful que guarda el *cart de compras* en memoria.

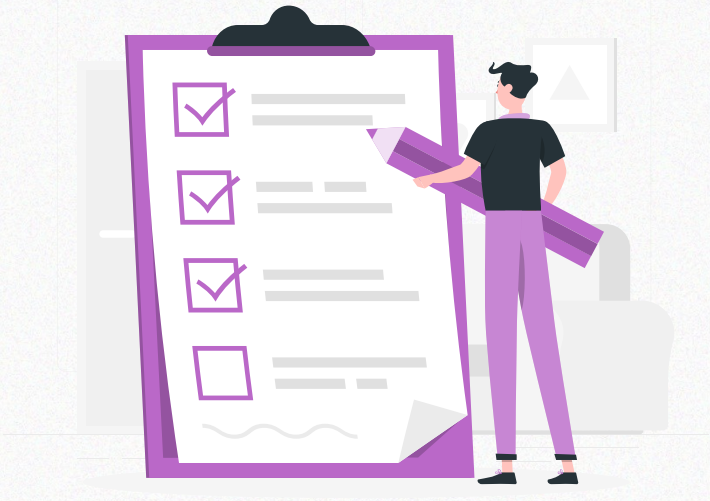
Problema: cuando el servicio escala horizontalmente (varias instancias), el cliente puede ser redirigido a otra instancia y perder su carrito.

External Configuration

Separar la configuración (variables de entorno, rutas de conexión, credenciales, etc.) del código es fundamental para la agilidad.

De este modo, la misma imagen (contenedor) puede usarse en diferentes entornos (desarrollo, pruebas, producción) simplemente modificando la configuración externa.

Se hace hincapié en **no** almacenar credenciales o claves sensibles en el repositorio de código, sino en un servicio de configuración (HashiCorp Vault, Kubernetes Secrets, etc.).



Infraestructura Inmutable, Escalabilidad y Resiliencia

Declarative Deployment

Los despliegues en *cloud-native* se benefician del enfoque declarativo, donde describimos “el estado deseado” de la infraestructura y los servicios (por ejemplo, usar un archivo YAML en Kubernetes o plantillas de CloudFormation/Terraform).

Es un contraste con el enfoque imperativo, donde se ejecutarían una serie de comandos manuales cada vez. El despliegue declarativo mejora la trazabilidad y repetibilidad.

Declarative Deployment

Kubernetes compara continuamente el estado real (cuántos contenedores están corriendo, qué versiones tienen, etc.) con el estado deseado (lo que declaraste en tu archivo YAML). Si detecta diferencias, ajusta la infraestructura para que coincida con lo que está declarado en el archivo.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mi-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mi-app
  template:
    metadata:
      labels:
        app: mi-app
    spec:
      containers:
      - name:
contenedor-app
        image: mi-
imagen:v1
        ports:
        - containerPort:
80
```

SaaS

Modelo de computación en la nube donde el software se ofrece como un servicio accesible desde un navegador u otra interfaz, generalmente con un modelo de suscripción.

¿Cómo funciona?

El proveedor aloja, mantiene y actualiza la aplicación; el usuario solo la consume a través de internet.

Ejemplos

- **Microsoft 365**
- **Salesforce**
- **Google Workspace**

PaaS

Modelo de computación en la nube que proporciona un entorno completo para desarrollar, probar y desplegar aplicaciones. Incluye sistemas operativos, bases de datos y herramientas.

¿Cómo funciona?

El proveedor administra la infraestructura y middleware; los desarrolladores se centran en el código y la lógica de negocio sin preocuparse por la configuración de servidores.

Ejemplos

- **Vercel**
- **Google App Engine**
- **Heroku**

iaaS

Modelo de computación en la nube que provee recursos virtualizados (servidores, redes y almacenamiento) bajo demanda.

¿Cómo funciona?

El proveedor se encarga del hardware y la virtualización; el usuario instala y administra el sistema operativo y las aplicaciones.

Ejemplos

- **Amazon Web Services (AWS) EC2**
- **Google Compute Engine**:
- **Microsoft Azure Virtual Machines**

Conociendo la Nube

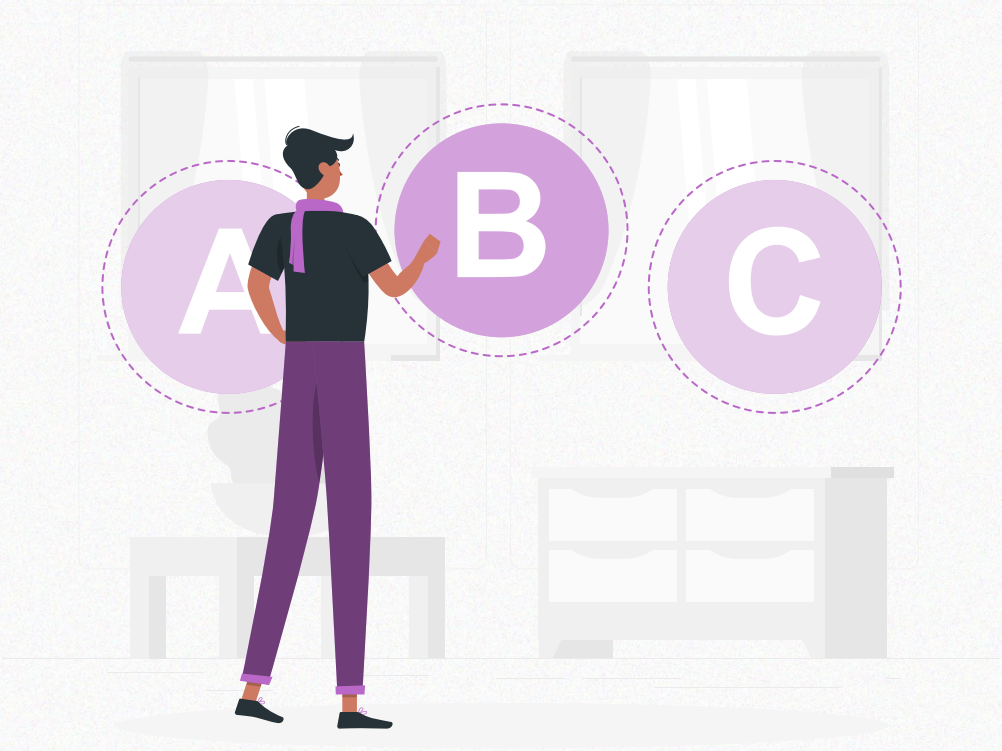
Tipos de implementación de la nube

Nube pública: Infraestructura propiedad de un proveedor externo (Amazon, Microsoft, Google, IBM, etc.).

Nube privada: Infraestructura dedicada de uso exclusivo para una sola organización.

Nube híbrida: Combina servicios de nube pública y privada, permitiendo que los datos y aplicaciones se muevan entre ambas.

Multicloud: Uso de múltiples proveedores de nube pública (AWS, Azure, Google Cloud, etc.) al mismo tiempo.



Contenedores y Orquitestacion

Un **contenedor** es una unidad estandarizada de software que empaqueta el código y todas sus dependencias para que una aplicación se ejecute de manera rápida y confiable de un entorno de cómputo a otro (Docker, 2013).

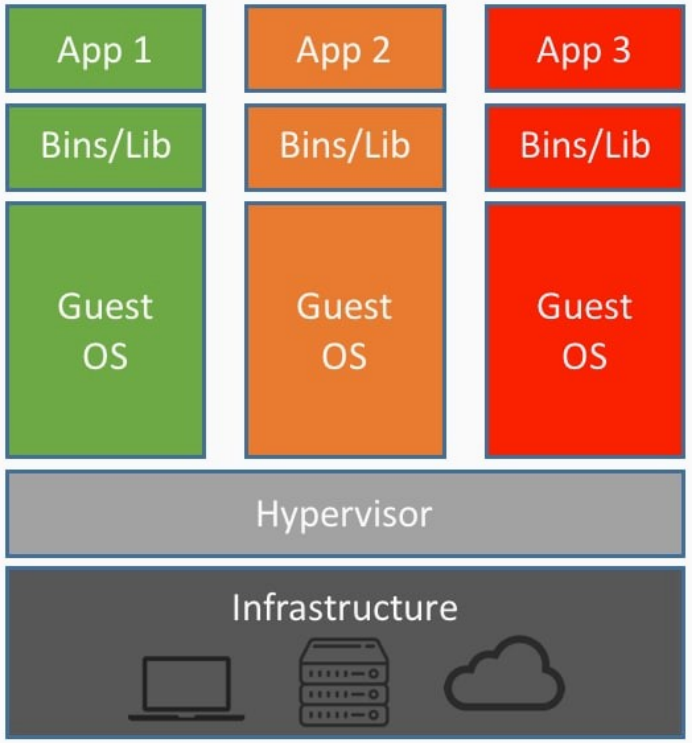
Los contenedores proporcionan una manera de encapsular la lógica de negocio y sus dependencias en un paquete reproducible y portable, facilitando la migración entre distintas nubes y entornos de desarrollo.

Contenedores y Orquitestacion

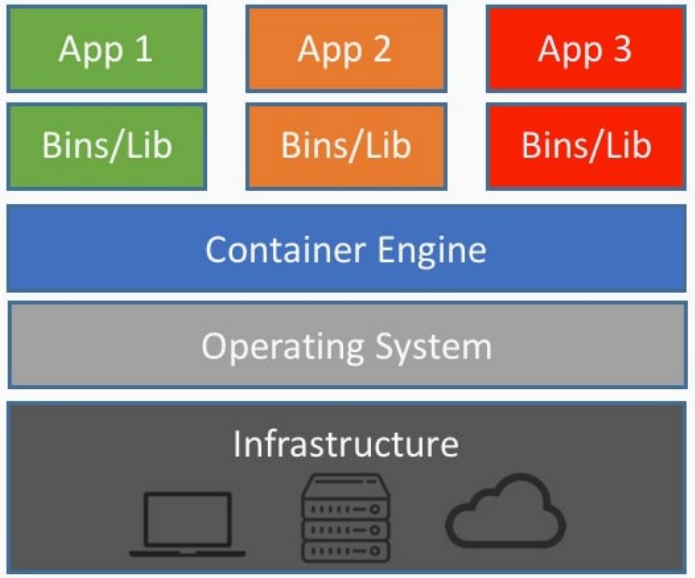
1. **Portabilidad:** al empaquetar la aplicación con todas sus librerías y configuraciones, el contenedor puede ejecutarse en prácticamente cualquier infraestructura (on-premise o nube).

2. **Aislamiento:** cada contenedor opera en un espacio aislado, evitando conflictos de dependencias con otros contenedores.

3. **Eficiencia:** comparten el kernel del SO, reduciendo la sobrecarga de recursos en comparación con las VMs.



Machine Virtualization



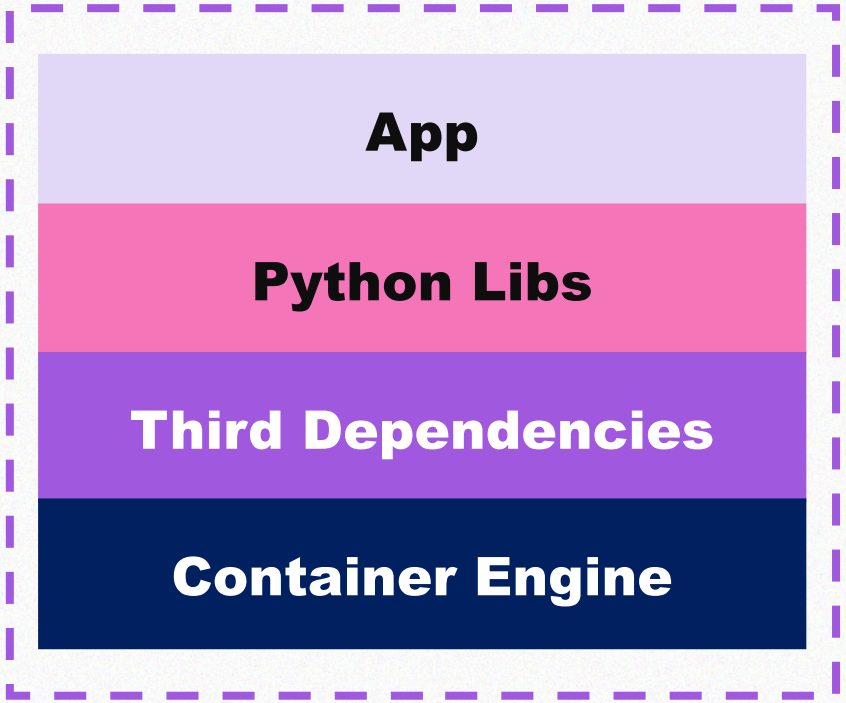
Containers

Contenedores y Orquitestacion

Ejemplo Breve

Supongamos que tenemos una aplicación escrita en Python que depende de varias librerías (requests, flask, etc.).

Al crear una **imagen Docker**, se incluye el runtime de Python y las librerías necesarias, de forma que al desplegar el contenedor en un servidor o en la nube, el entorno de ejecución ya está preconfigurado y garantizado.



Docker

Docker es una **plataforma** que permite a los desarrolladores crear, desplegar y ejecutar aplicaciones dentro de **contenedores**. Los contenedores son entornos ligeros y portátiles que contienen todo lo necesario para que una aplicación funcione: código, runtime, bibliotecas y configuraciones.

Ventajas de Usar Docker

- Portabilidad
- Consistencia entre Entornos
- Aislamiento de Aplicaciones
- Eficiencia en Recursos

Dockerfile

Plantillas inmutables que contienen todo lo necesario para ejecutar una aplicación.

```
FROM elixir:1.14-alpine

# Fijamos la variable de entorno para producción
ENV MIX_ENV=prod

# Creamos un directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiamos los archivos del proyecto al contenedor
COPY . /app

# Instalamos dependencias y compilamos el proyecto
RUN mix local.hex --force && \
mix local.rebar --force && \
mix deps.get && \
mix compile

# Ejecutamos la aplicación al iniciar el contenedor
CMD ["mix", "run", "-e", "DemoElixir.CLI.start()"]
```

Estructura Básica de un Dockerfile

Un Dockerfile está compuesto por una secuencia de **instrucciones**. Cada instrucción crea una capa en la imagen final.

1. **FROM:** Especifica la imagen base.
2. **LABEL:** Agrega metadatos a la imagen.
3. **RUN:** Ejecuta comandos en la imagen durante la construcción.
4. **COPY:** Copia archivos y directorios desde el contexto de construcción al sistema de archivos de la imagen.
5. **ADD:** Similar a COPY pero con funcionalidades adicionales (como extraer archivos comprimidos).
6. **WORKDIR:** Establece el directorio de trabajo.
7. **ENV:** Define variables de entorno.
8. **EXPOSE:** Indica los puertos que la aplicación utilizará.
9. **CMD:** Especifica el comando por defecto a ejecutar al iniciar un contenedor.
10. **ENTRYPOINT:** Define el punto de entrada de la aplicación.
11. **VOLUME:** Crea puntos de montaje para volúmenes.
12. **USER:** Especifica el usuario que ejecutará las instrucciones posteriores.

Volúmenes y Redes

Volúmenes:

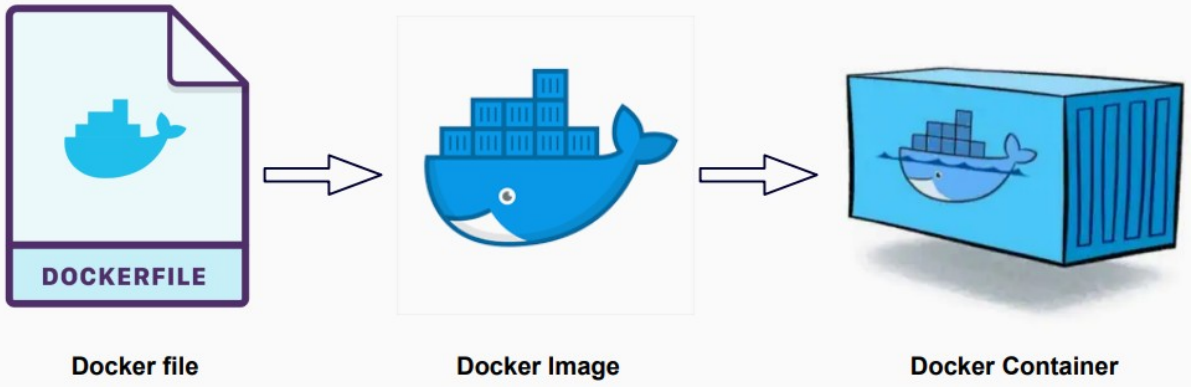
- **Propósito:** Almacenamiento persistente que persiste más allá del ciclo de vida de un contenedor.
- **Uso Común:** Bases de datos, archivos de configuración.
- **Comando de Creación:** `docker volume create my_volume`

Redes:

- **Propósito:** Facilitar la comunicación entre contenedores.
- **Tipos:** Bridge (predeterminado), Host, Overlay.
- **Comando de Creación:** `docker network create my_network`

Construir una imagen Docker

```
docker build -t mi-imagen .
```



Iniciar un Contenedor

```
docker run -d -p 8080:80 --name mi-contenedor mi-imagen
```

- `-d`: Ejecuta el contenedor en segundo plano (detached).
- `-p`: Mapea puertos del host al contenedor.
- `--name`: Asigna un nombre al contenedor.

Kubernetes: El orquestador de contenedores

Kubernetes es un sistema de orquestación de contenedores de código abierto que automatiza la implementación, el escalado y la administración de aplicaciones en contenedores.

El proyecto fue originalmente desarrollado por Google y posteriormente donado a la Cloud Native Computing Foundation (CNCF). Kubernetes se encarga de coordinar múltiples contenedores distribuidos en clústeres de servidores, asegurando alta disponibilidad, escalabilidad y resiliencia (Burns et al., 2016).

Componentes Principales de Kubernetes

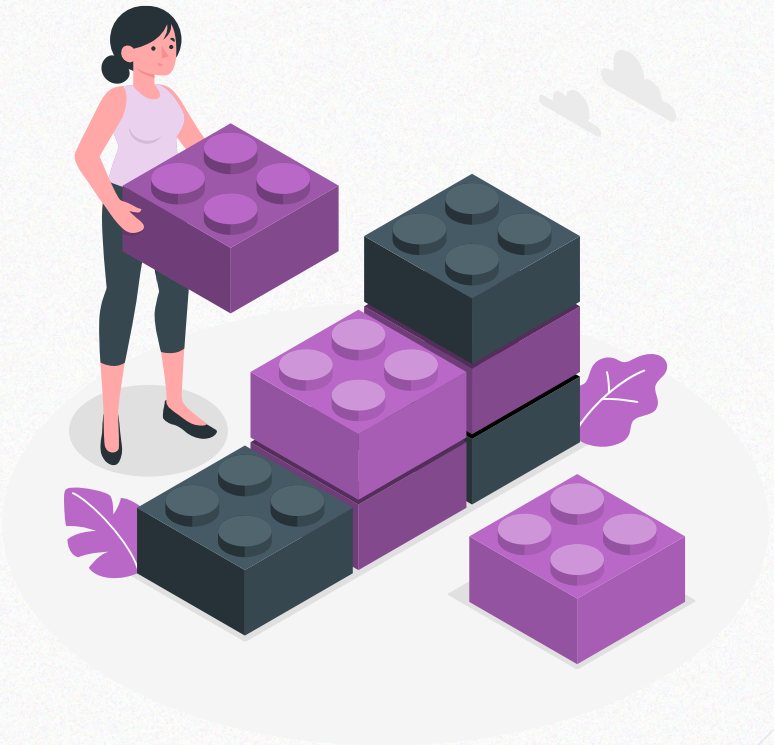
1. **Pod**: La unidad mínima de despliegue en Kubernetes. Un pod puede contener uno o varios contenedores que comparten el mismo espacio de red y almacenamiento.

2. **Deployment**: Define el estado deseado de la aplicación (número de réplicas, la imagen de contenedor a utilizar, etc.) y lo mantiene de forma declarativa.

3. **Service**: Actúa como punto estable de red para un conjunto de pods, permitiendo el *service discovery* y el balanceo de carga básico.

4. **ConfigMap y Secret**: Mecanismos para inyectar configuraciones y credenciales sensibles a los contenedores sin tener que modificar la imagen.

5. **Controller Manager, Scheduler y etcd**: Componentes internos de Kubernetes que gestionan el estado del clúster, asignan recursos y almacenan la información de forma distribuida.



Ventajas de Kubernetes

- **Escalabilidad Horizontal Automática (Horizontal Pod Autoscaler)**: ajusta dinámicamente el número de pods en función de métricas como uso de CPU o memoria.
- **Autocuración (Self-Healing)**: reinicia contenedores que fallan, los reemplaza, elimina contenedores que no responden a los checks de salud, etc.
- **Despliegues Continuos**: facilita procesos como *rolling updates* y *rollbacks*, minimizando el tiempo de inactividad de la aplicación.
- **Ecosistema Extensible**: a través de *operators*, *helm charts* y *custom resource definitions (CRDs)* se pueden automatizar tareas más especializadas.

Ventajas de Kubernetes

- **Escalabilidad Horizontal Automática (Horizontal Pod Autoscaler)**: ajusta dinámicamente el número de pods en función de métricas como uso de CPU o memoria.
- **Autocuración (Self-Healing)**: reinicia contenedores que fallan, los reemplaza, elimina contenedores que no responden a los checks de salud, etc.
- **Despliegues Continuos**: facilita procesos como *rolling updates* y *rollbacks*, minimizando el tiempo de inactividad de la aplicación.
- **Ecosistema Extensible**: a través de *operators*, *helm charts* y *custom resource definitions (CRDs)* se pueden automatizar tareas más especializadas.

Pod

El **Pod** es la unidad más pequeña en Kubernetes. Es un contenedor lógico que puede contener uno o más contenedores (aunque en la mayoría de los casos, un Pod contiene un solo contenedor).

Todos los contenedores dentro de un Pod:

- Comparten la misma dirección IP.
- Pueden comunicarse entre sí usando `localhost`.
- Comparten volúmenes para el almacenamiento de datos.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
    ports:
    - containerPort: 80
```

Deployment

Un **Deployment** es un recurso de Kubernetes que administra la creación y el escalado de los Pods. Garantiza que siempre haya el número deseado de Pods en ejecución.

Características:

- Si un Pod falla, Kubernetes lo reemplaza automáticamente.
- Permite realizar actualizaciones sin interrupciones (rolling updates).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx
        ports:
        - containerPort: 80
```

Service

Un **Service** expone un conjunto de Pods como un servicio de red. Esto permite que los Pods se comuniquen entre sí o que las aplicaciones externas accedan a los Pods.

Tipos de Services:

1. **ClusterIP:** Expone el servicio dentro del clúster.
2. **NodePort:** Expone el servicio en un puerto específico en cada nodo.
3. **LoadBalancer:** Proporciona una IP externa para el servicio.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

ConfigMap

Un **ConfigMap** permite almacenar configuraciones en formato de clave-valor y compartirlas con los Pods.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  config-key: "config-value"
```

Secret

Un **Secret** almacena datos sensibles como credenciales o claves de acceso. Es similar a un ConfigMap, pero los datos están encriptados.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4= # Base64 de "admin"
  password: cGFzc3dvcmQ= # Base64 de "passwor
```

ConfigMap

Un **ConfigMap** permite almacenar configuraciones en formato de clave-valor y compartirlas con los Pods.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  config-key: "config-value"
```

Secret

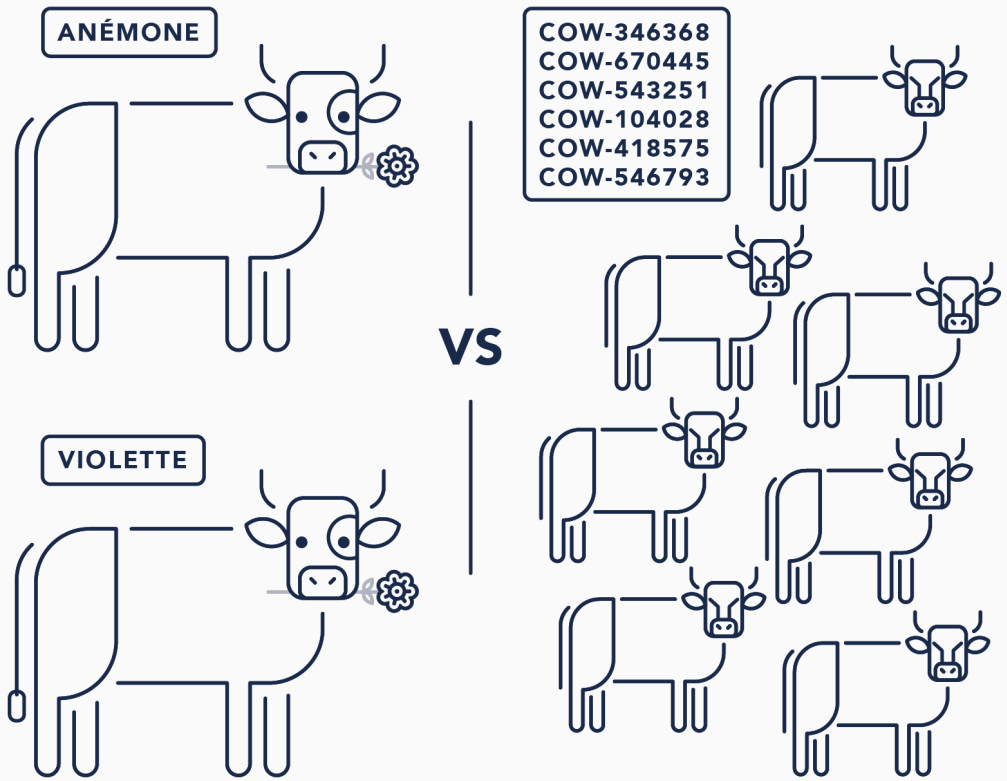
Un **Secret** almacena datos sensibles como credenciales o claves de acceso. Es similar a un ConfigMap, pero los datos están encriptados.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4= # Base64 de "admin"
  password: cGFzc3dvcmQ= # Base64 de "passwor
```

Infraestructura Inmutable

Se refiere al despliegue de entornos donde los servidores o contenedores no se “parchean” ni modifican directamente, sino que se reemplazan por versiones nuevas.

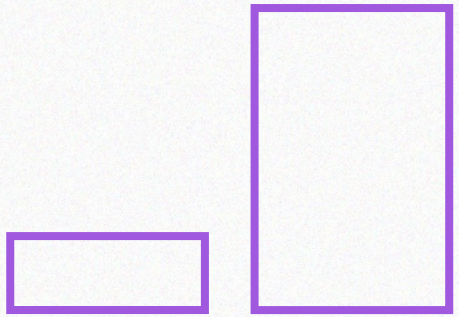
Alude al principio “*Pets vs. Cattle*”: tratar los servidores como “ganado” intercambiable y no como “mascotas” únicas (Davis, 2019).



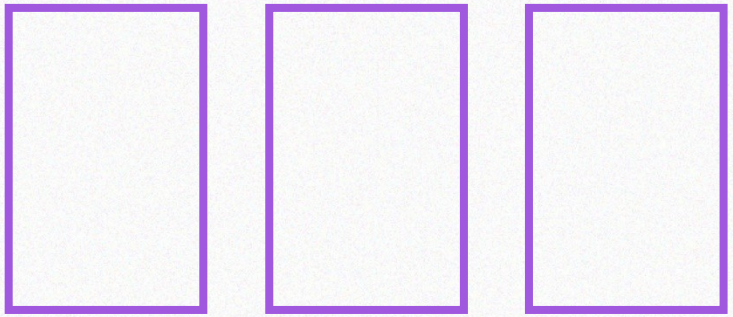
Escalabilidad

- **Horizontal**: Añadir más instancias (contenedores, pods, máquinas virtuales).
- **Vertical**: Aumentar la capacidad de una instancia (más memoria, más CPU).

En entornos cloud native se prioriza la escalabilidad **horizontal**, aprovechando orquestadores como Kubernetes (Ibryam & Huss, 2019).



Escalamiento Vertical

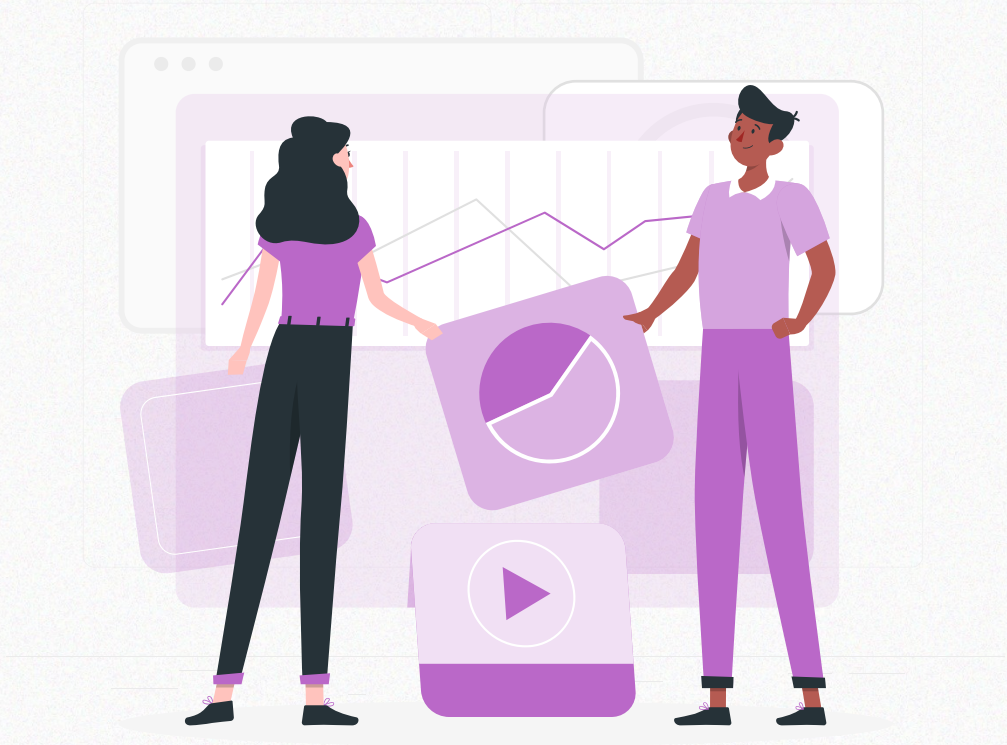


Escalamiento Horizontal

Resiliencia

Capacidad de un sistema para “recuperarse” de fallos y seguir operando.

Incluye la adopción de patrones como *circuit breakers*, *reintentos* y *timeouts* (Davis, 2019; Indrasiri & Siriwardena, 2021).



Patrones de Kubernetes (basado en *Kubernetes Patterns*)

***Sidecar Pattern*:** Separar responsabilidades en contenedores auxiliares (logging, proxy, etc.).

***Ambassador Pattern*:** Cómo exponer servicios a través de contenedores intermedios (proxys).

***Adapter Pattern*:** Transformar datos y protocolos entre sistemas.

¿Para qué creen que sirve?

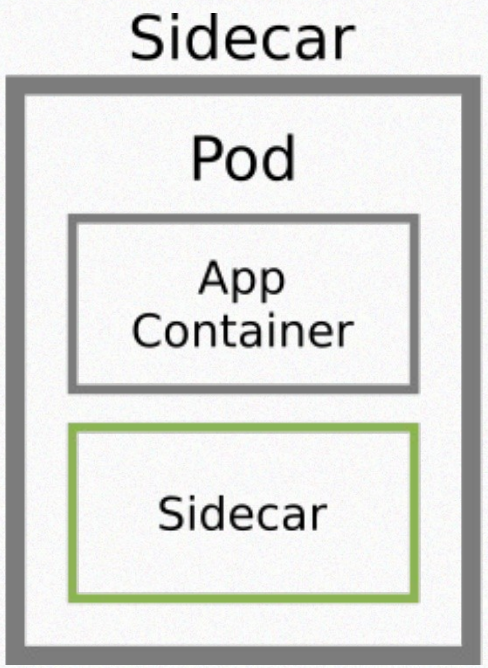
Supongamos que tienes una aplicación muy simple que procesa archivos.

¿Qué pasa si ahora quieres agregarle logging avanzado o métricas Prometheus?

¿Modificarías el código directamente o hay otra opción?

Si no puedes tocar el código de la app (es un binario cerrado)

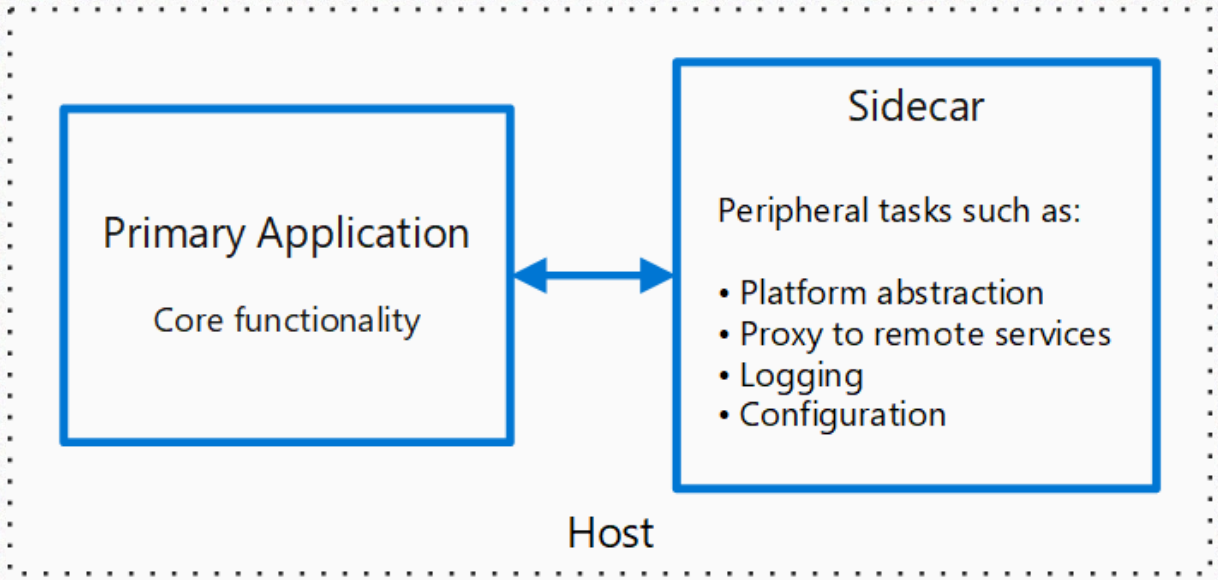
¿cómo podrías agregarle estas capacidades?



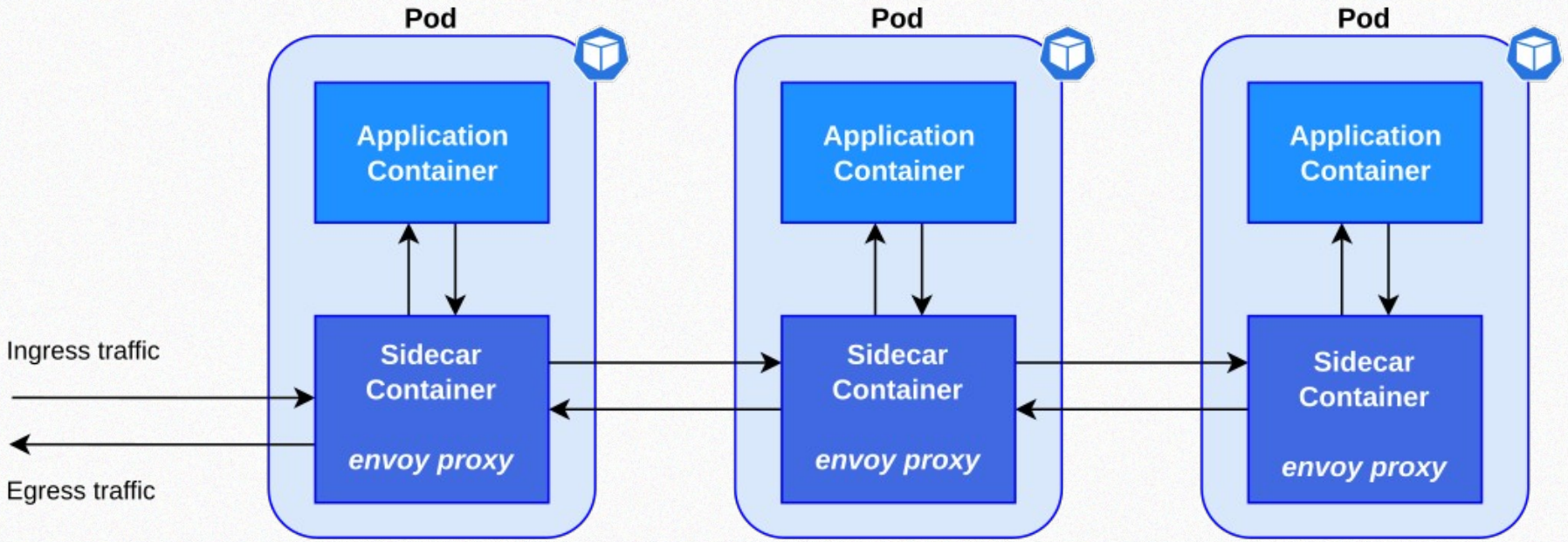
Patrón Sidecar

El patrón **Sidecar** implica ejecutar contenedores auxiliares (sidecars) junto al contenedor principal de una aplicación dentro de un mismo *pod* en Kubernetes.

Estos contenedores secundarios asumen responsabilidades complementarias, como logging, monitoreo, proxies, entre otros, separando así las preocupaciones y manteniendo el contenedor principal enfocado en la lógica de negocio.

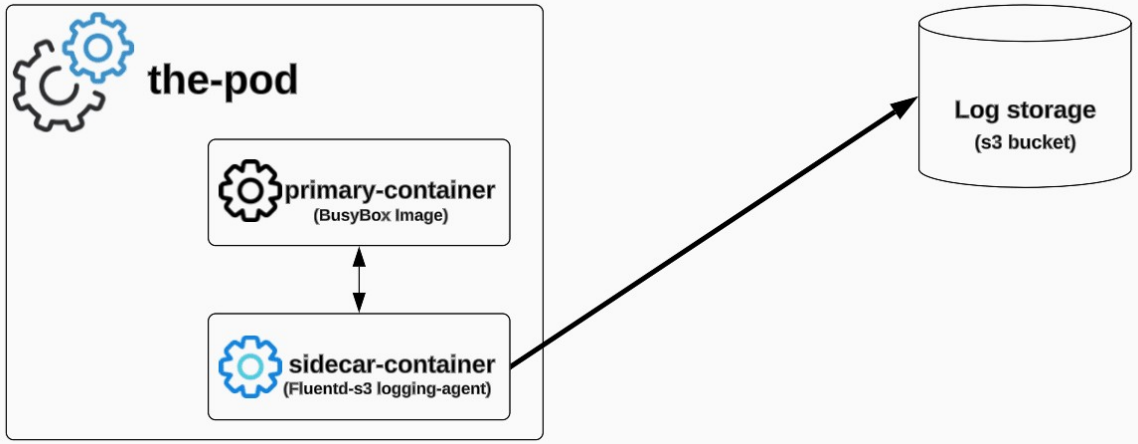


Patrón Sidecar



Patrón Sidecar

- **Separación de responsabilidades**: Facilita la mantenibilidad y escalabilidad.
- **Reutilización**: Los sidecars pueden ser reutilizados por múltiples aplicaciones.
- **Independencia**: Permite actualizar o reiniciar el sidecar sin afectar al contenedor principal.




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-with-sidecar
spec:
  containers:
    - name: app-container
      image: mi-aplicacion:latest
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: logs
          mountPath: /var/log/app
    - name: fluentd-sidecar
      image: fluentd:latest
      env:
        - name: FLUENTD_CONF
          value: fluentd.conf
      volumeMounts:
        - name: logs
          mountPath: /var/log/app
  volumes:
    - name: logs
      emptyDir: {}
```

¿Alguna vez han necesitado un traductor o representante que hable con alguien por ustedes porque no dominan su idioma o no pueden exponerse directamente?

En arquitectura de software, existe un patrón llamado ***Ambassador****, que juega un rol muy similar para las aplicaciones.*

Patrón Ambassador

El patrón **Ambassador** utiliza un contenedor intermedio (proxy) dentro del mismo *pod* que actúa como intermediario entre la aplicación y el mundo exterior. Este proxy maneja tareas como enrutamiento de tráfico, balanceo de carga, autenticación, y más, permitiendo que la aplicación principal se concentre en su funcionalidad principal.

- **Flexibilidad en el enrutamiento**: Facilita la gestión del tráfico entrante y saliente.
- **Seguridad**: Permite implementar medidas de seguridad a nivel de proxy.
- **Observabilidad**: Facilita la monitorización y trazabilidad del tráfico.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-with-ambassador
spec:
  containers:
    - name: app-container
      image: mi-aplicacion:latest
      ports:
        - containerPort: 8080
    - name: envoy-ambassador
      image: envoyproxy/envoy:v1.18.3
      ports:
        - containerPort: 9901 # Admin port
        - containerPort: 10000 # Proxy port
      volumeMounts:
        - name: envoy-config
          mountPath: /etc/envoy
      command: ["envoy", "-c",
"/etc/envoy/envoy.yaml"]
  volumes:
    - name: envoy-config
      configMap:
        name: envoy-config
```

Patrón Ambassador

¿Qué tareas creen que podría asumir un intermediario entre la app y un servicio externo?

- Traducción de protocolos (HTTP ↔ gRPC).
- Manejo de autenticación con el servicio externo.
- Retransmisión de peticiones y balanceo de carga.

Tienes una app que debe comunicarse con un servicio REST en https://api.externa.com, pero no puede hacer conexiones TLS directamente.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-with-ambassador
spec:
  containers:
  - name: web-app
    image: mycompany/web-app:1.0
    env:
    - name: API_URL
      value: "http://localhost:8080"
  - name: ambassador
    image: envoyproxy/envoy:v1.21
    args:
    - --config-path /etc/envoy/envoy.yaml
    ports:
    - containerPort: 8080
```

Patrón Ambassador

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: envoy-config
data:
  envoy.yaml: |
    static_resources:
      listeners:
        - name: listener_0
          address:
            socket_address:
              address: 0.0.0.0
              port_value: 10000
          filter_chains:
            - filters:
                - name: envoy.filters.network.http_connection_manager
                  typed_config:
                    "@type": type.googleapis.com/envoy.extensions.filters.network.http_conne
                    stat_prefix: ingress_http
                    route_config:
                      name: local_route
                      virtual_hosts:
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: app-service
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 10000
  selector:
    app: app-with-ambassador
```

Patrón Adapter

El patrón **Adapter** actúa como un puente que permite la interacción entre sistemas que, de otro modo, serían incompatibles debido a diferencias en datos o protocolos. En Kubernetes, un contenedor adaptador puede transformar datos, protocolos o interfaces para que distintos servicios puedan comunicarse eficazmente.

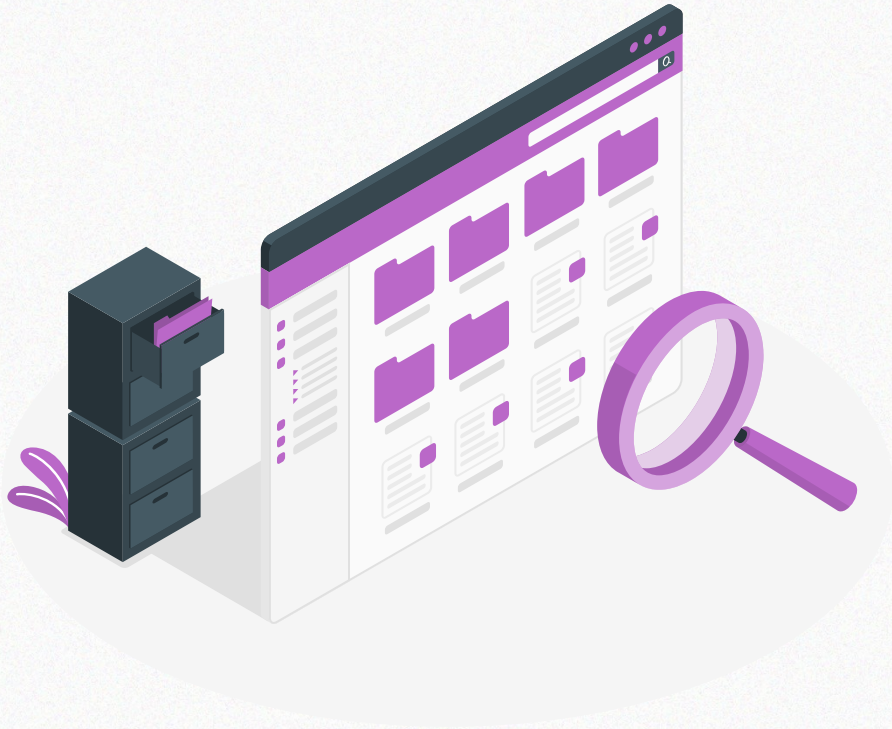
- **Compatibilidad**: Facilita la integración de servicios heterogéneos.
- **Flexibilidad**: Permite la evolución independiente de sistemas conectados.
- **Reutilización**: Los adaptadores pueden ser reutilizados para múltiples servicios.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-with-adapter
  labels:
    app: app-with-adapter
spec:
  containers:
    - name: main-app
      image: mi-aplicacion:latest
      ports:
        - containerPort: 8080
      env:
        - name: ADAPTER_URL
          value: http://localhost:5000/conv
    - name: adapter-container
      image: mi-adaptador:latest
      ports:
        - containerPort: 5000
```

Patrones de Datos, Eventos y Funciones

En el desarrollo de aplicaciones modernas en la nube, adoptar **patrones de diseño** adecuados es esencial para garantizar la escalabilidad, resiliencia y mantenibilidad.

1. **Datos en la Nube**
2. **Arquitecturas Basadas en Eventos**
3. **Funciones como Servicio (Serverless)**



¿Si apagas y prendes un servidor en la nube, deberían perderse los datos?

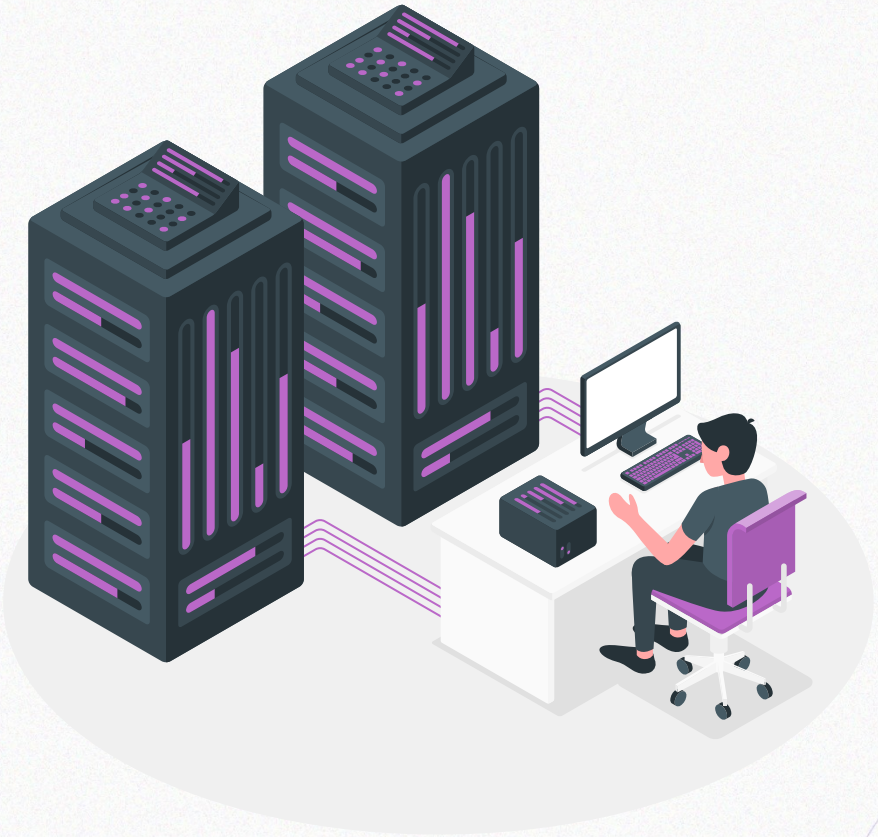
¿Qué pasa si tu app se mueve de un nodo a otro en Kubernetes?

Datos en la Nube
Almacenamiento Distribuido vs. Almacenamiento Acoplado

Almacenamiento Acoplado

El almacenamiento acoplado está estrechamente integrado con los recursos de cómputo, generalmente refiriéndose a almacenamiento local o centralizado no distribuido ampliamente.

- No sobrevive al reinicio o reprogramación de Pods.
- No soporta múltiples instancias concurrentes.
- No es portable entre regiones o zonas de disponibilidad.



Datos en la Nube
Almacenamiento Distribuido vs. Almacenamiento Acoplado

Almacenamiento Distribuido

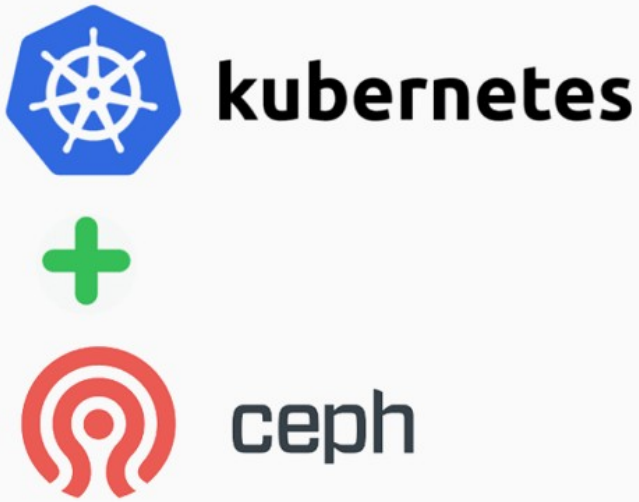
El almacenamiento distribuido dispersa datos a través de múltiples nodos o ubicaciones físicas, mejorando la disponibilidad, escalabilidad y tolerancia a fallos.

- Datos persistentes aunque el Pod se destruya.
- Acceso concurrente desde múltiples instancias.
- Soporta escalamiento horizontal y failover automático.

Esto se conecta con el **12 Factor App**:

- **Factor 6: Processes are stateless and share-nothing**
- **Factor 9: Disposability**

Almacenamiento Distribuido




```yaml
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: replicapool
  namespace: rook-ceph
spec:
  replicated:
    size: 3
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ceph-pvc
spec:
  storageClassName: rook-ceph-block
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Almacenamiento Acoplado

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nfs-pod
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - mountPath:
"/usr/share/nginx/html"
          name: nfs-storage
  volumes:
    - name: nfs-storage
      persistentVolumeClaim:
        claimName: nfs-pvc
```

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  nfs:
    path: /exported/path
    server: 192.168.1.100
```

Storage en Arquitecturas Basadas en Eventos

¿Qué pasa cuando compras un producto en Amazon?

¿Creen que todos los sistemas *(inventario, facturación, envío)* se actualizan en un solo instante?

En arquitecturas basadas en eventos no ocurre todo al mismo tiempo. Aquí el almacenamiento es más complejo que un simple INSERT INTO en una tabla SQL.

Patrones de almacenamiento en Event-Driven

Event Store (Event Sourcing)

- Guarda cada cambio como un evento inmutable.
- El estado actual se reconstruye aplicando todos los eventos en orden.
- *Ventaja:* Auditable, reproduce el sistema en cualquier punto.
- *Desventaja:* Complejidad al consultar datos.

CQRS (Command Query Responsibility Segregation)

- Separa el almacenamiento de comandos (*writes*) del de consultas (*reads*).
- *Ventaja:* Escala lecturas y escrituras de forma independiente.
- *Desventaja:* Mayor coordinación de datos.

Patrones de almacenamiento en Event-Driven

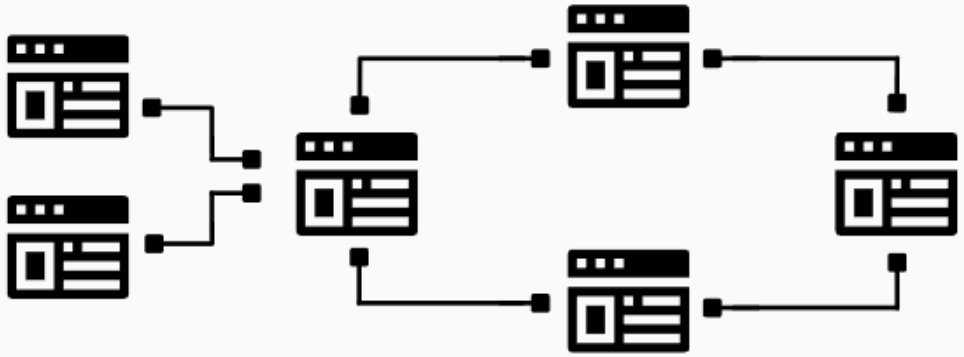
Log-Based Storage (Kafka, Pulsar)

- Usa un *log* como el sistema de almacenamiento principal.
- *Ventaja:* Escalable y distribuido.
- *Desventaja:* Manejar la retención de datos.

State Store Local (Caching + Snapshotting)

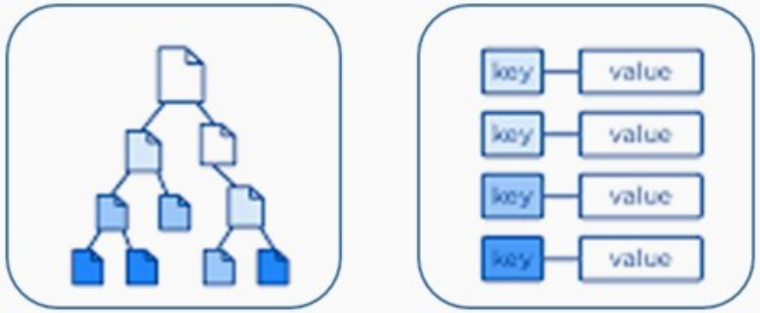
- Servicios mantienen su propio estado derivado de eventos.
- *Ventaja:* Desacoplamiento fuerte.
- *Desventaja:* Posibles inconsistencias temporales.

Bases de Datos Relacionales



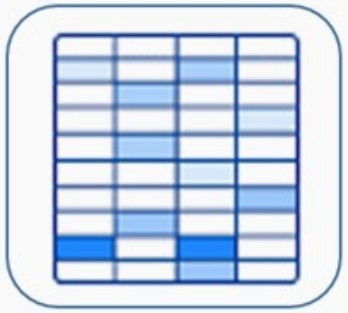
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:13
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_DB
              value: "mi_base_de_datos"
            - name: POSTGRES_USER
              value: "usuario"
            - name: POSTGRES_PASSWORD
              value: "contraseña_segura"
          volumeMounts:
            - mountPath:
```

Bases de Datos No Relacionales

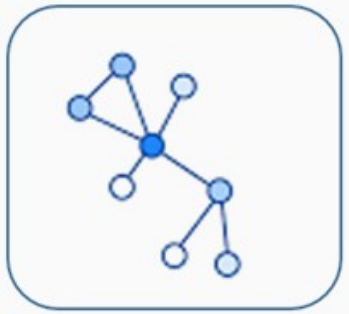


Document Store

Key-Value Store



Wide-Column Store



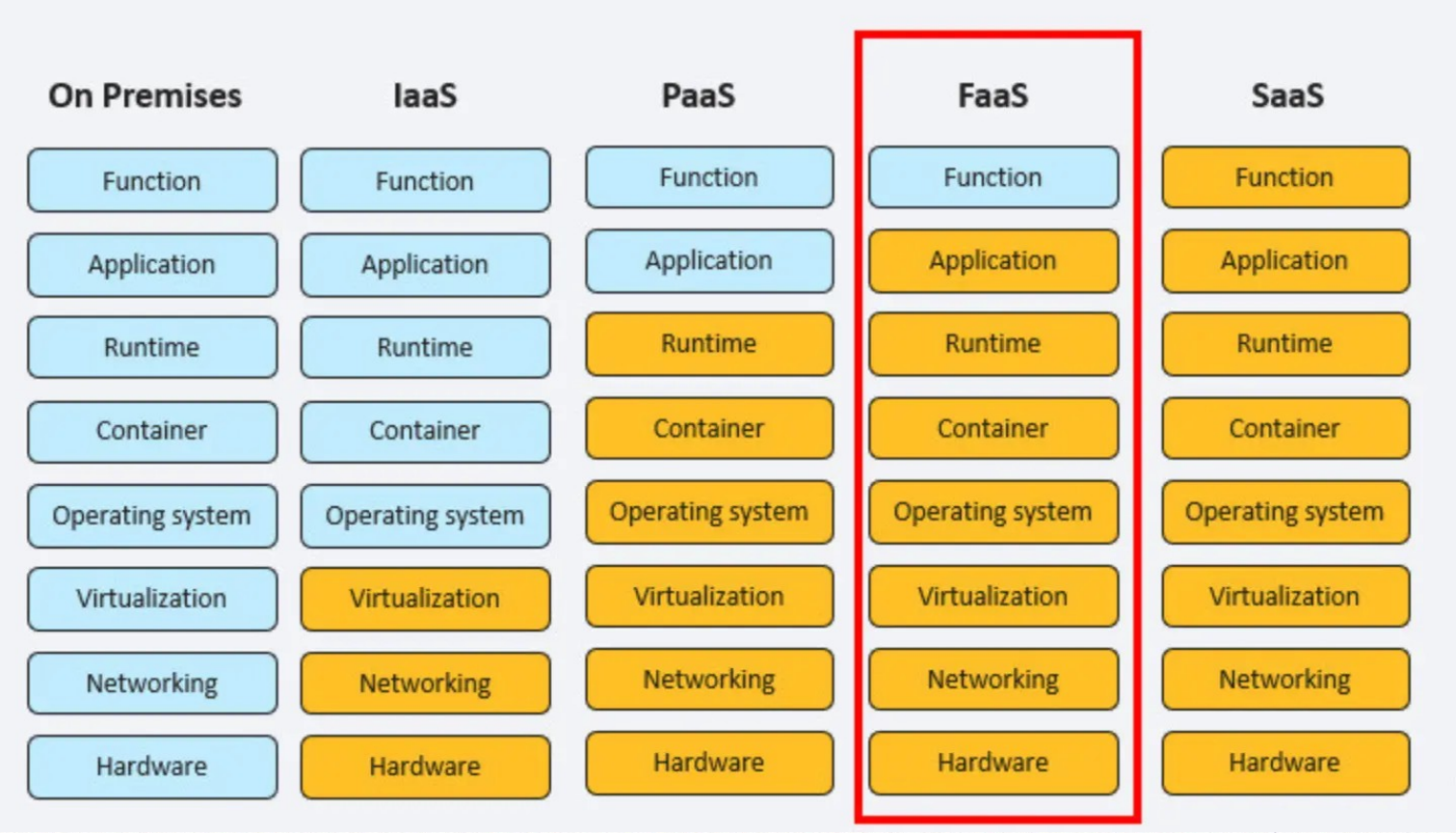
Graph Store

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mongodb
spec:
  ports:
    - port: 27017
  selector:
    app.kubernetes.io/name: mongodb
```

Funciones como Servicio (FaaS)

La FaaS, o función como servicio, es un servicio de cloud computing que permite a los clientes ejecutar código en respuesta a sucesos, sin gestionar la compleja infraestructura asociada normalmente a la creación y lanzamiento de aplicaciones de microservicios.

A menudo se confunden las funciones como servicio (FaaS) y sin servidor, pero lo cierto es que la FaaS es en realidad un subconjunto de las funciones sin servidor.



FaaS

*Es como **usar Uber***
No te preocupas por comprar, mantener o estacionar el auto. Solo lo llamas cuando lo necesitas, te lleva a tu destino, y pagas por ese viaje específico.

Cómo funciona FaaS

1. Subes tu función (código + dependencias).
2. Se define un trigger (evento que dispara la función).
3. La nube crea un contenedor efímero, ejecuta la función y luego lo apaga.
4. Pagas solo por los milisegundos en que la función estuvo corriendo.

Proveedor	Servicio FaaS	Lenguajes soportados
AWS	Lambda	Python, Node.js, Java, Go
Google Cloud	Cloud Functions	Python, Node.js, Go
Azure	Azure Functions	C#, JavaScript, Python
Open Source	OpenFaaS, Knative, Kubeless	Cualquiera con contenedores

Funciones como Servicio (FaaS)

```
/**
 * Responds to any HTTP request.
 *
 * @param {Object} req Cloud Function request context.
 * @param {Object} res Cloud Function response context.
 */
exports.saludar = (req, res) => {
  const nombre = req.query.nombre || req.body.nombre || 'Mundo';
  res.send(`¡Hola, ${nombre}!`);
};
```

Funciones como Servicio (FaaS)

```
{
  "name": "mi-funcion-faas",
  "version": "1.0.0",
  "dependencies": {}
}
```

```
gcloud auth login
gcloud config set project TU_ID_DE_PROYECTO
```

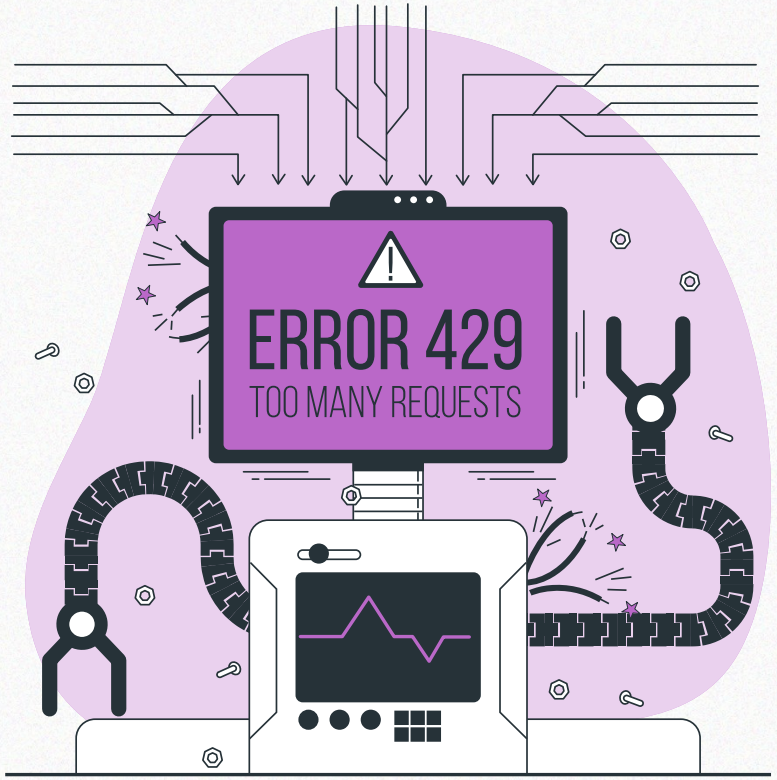
Funciones como Servicio (FaaS)

```
exports.apiTareas = async (req, res) => {
  const collection = firestore.collection('tareas');
  const { method, query, body } = req;

  try {
    switch (method) {
      case 'GET':
        if (query.id) {
          // Obtener una tarea específica
          const doc = await collection.doc(query.id).get();
          if (!doc.exists) {
            res.status(404).send('Tarea no encontrada');
          } else {
            res.status(200).json(doc.data());
          }
        } else {
          // Obtener todas las tareas
          const snapshot = await collection.get();
          const tareas = [];
          snapshot.forEach(doc => {
            tareas.push({ id: doc.id, ...doc.data() });
          });
          res.status(200).json(tareas);
        }
        break;

      case 'POST':
        // Crear una nueva tarea
        const nuevaTarea = await collection.add(body);
        res.status(201).json({ id: nuevaTarea.id, ...body });
        break;
```

Diseño para Resiliencia y Cambio

Principio de Design for Failure: Resiliencia desde la Concepción

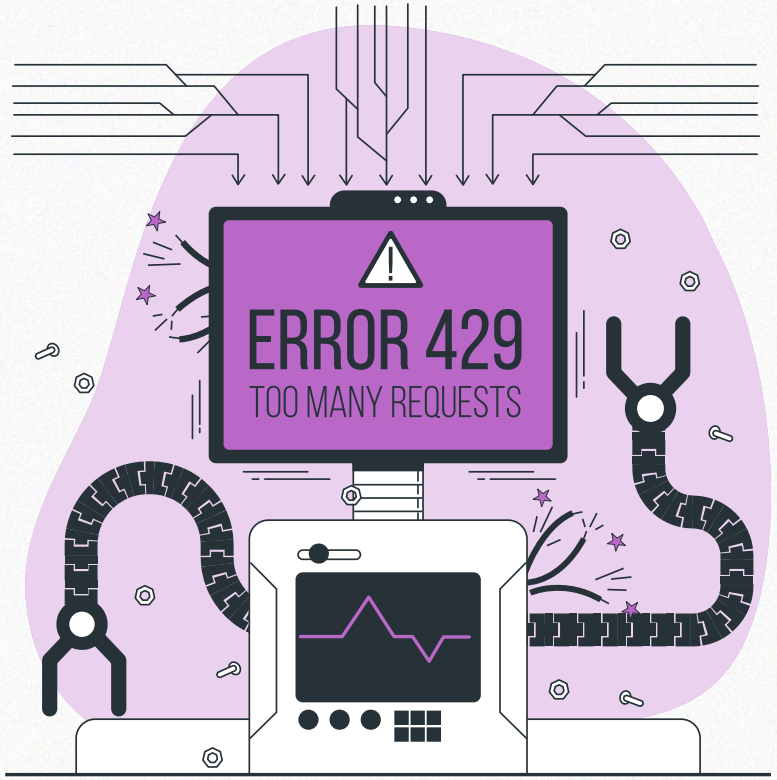
El principio de “Design for Failure” (Diseño para el Fallo) se basa en la idea de que los fallos son inevitables en sistemas distribuidos. En lugar de intentar prevenir todos los posibles fallos, se enfoca en diseñar sistemas que puedan recuperarse rápidamente y continuar funcionando incluso cuando ocurren errores.



Principio de Design for Failure: Resiliencia desde la Concepción

Conceptos Clave:

- **Redundancia:** Implementar múltiples instancias de componentes críticos para evitar puntos únicos de fallo.
- **Failover Automático:** Cambio automático a un sistema de respaldo en caso de fallo.
- **Circuit Breakers:** Patrón que previene que fallos en un servicio se propaguen a otros servicios.
- **Retry Logic:** Reintentar operaciones fallidas con estrategias de retroceso exponencial.



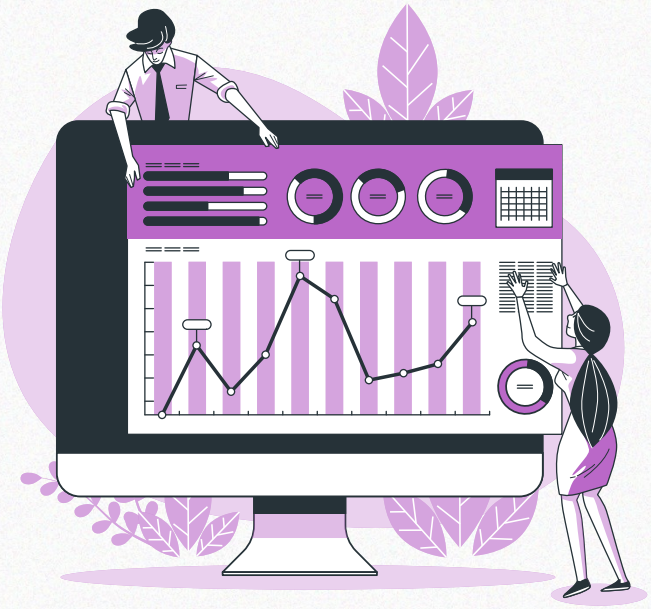
Principio de Design for Failure: Resiliencia desde la Concepción

Supongamos que tienes una aplicación web desplegada en Google Cloud que utiliza una base de datos SQL.

1. **Redundancia:** Configura instancias múltiples de tu aplicación detrás de un balanceador de carga para distribuir el tráfico.
2. **Failover Automático:** Configura réplicas de la base de datos en diferentes zonas de disponibilidad. Si la instancia principal falla, el sistema cambia automáticamente a una réplica.
3. **Circuit Breakers:** Implementa circuit breakers en tu aplicación para evitar que múltiples solicitudes fallen si la base de datos está inactiva.
4. **Retry Logic:** Configura reintentos automáticos para las solicitudes fallidas con tiempos de espera incrementales.

Monitoreo, Logging, Tracing (Patrones de Observabilidad)

La observabilidad es crucial para entender el comportamiento de las aplicaciones y detectar problemas rápidamente. Incluye monitoreo, logging y tracing para proporcionar una visión completa del sistema.



Componentes de la Observabilidad:

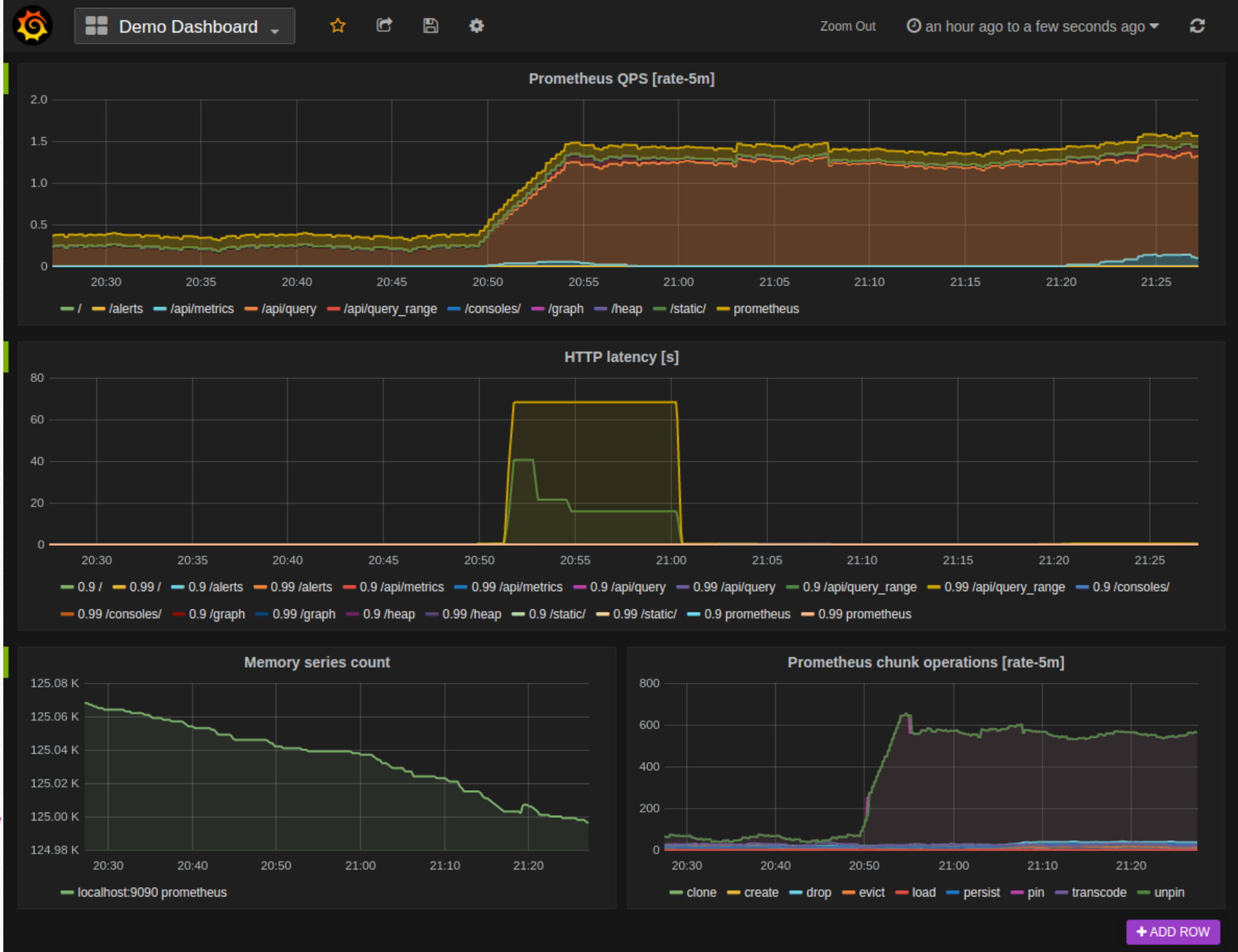
- **Monitoreo:** Recopilación y análisis de métricas en tiempo real (CPU, memoria, latencia, etc.).
- **Logging:** Registro de eventos y errores que ocurren dentro de la aplicación.
- **Tracing:** Seguimiento de solicitudes a través de múltiples servicios para identificar cuellos de botella y problemas de rendimiento.

Monitoreo, Logging, Tracing (Patrones de Observabilidad)

```
const express = require('express');
const client = require('prom-client');

const app = express();
const collectDefaultMetrics = client.collectDefaultMetrics;
collectDefaultMetrics();

app.get('/metrics', async (req, res) => {
  res.set('Content-Type', client.register.contentType);
  res.end(await client.register.metrics());
});

// Resto de tus rutas
app.listen(3000, () => {
  console.log('Aplicación escuchando en el puerto 3000');
});
```

Prometheus

Monitoreo, Logging, Tracing (Patrones de Observabilidad)



CODIGO
powered by Tecsup/>

Prometheus

new relic

Grafana

OpenTelemetry

Adaptabilidad y Tolerancia al Cambio

Immutable Infrastructure
La Infraestructura Inmutable es un enfoque en el que los servidores o componentes de infraestructura no se modifican una vez que han sido desplegados.

Blue-Green Deployments
El método de Despliegues Blue-Green implica mantener dos entornos de producción idénticos pero separados, denominados “Blue” y “Green”. Solo uno de estos entornos está activo y sirve el tráfico de usuarios en un momento dado.

Releases Canary (Canary Releases)
Las Releases Canary son una estrategia de despliegue progresivo donde una nueva versión de la aplicación se lanza inicialmente a un pequeño subconjunto de usuarios antes de desplegarla a toda la base de usuarios. El término “canario” hace referencia a la práctica minera de utilizar canarios para detectar gases tóxicos en minas.

Infraestructura Inmutable (Immutable Infrastructure)

La Infraestructura Inmutable es un enfoque en el que los servidores o componentes de infraestructura no se modifican una vez que han sido desplegados. En lugar de actualizar o parchear servidores existentes, se reemplazan completamente por nuevas versiones.

- **Reemplazo Completo:** En lugar de realizar cambios en los servidores actuales, se crea una nueva instancia con la configuración actualizada.
- **Consistencia:** Al crear nuevas instancias a partir de una imagen conocida, se asegura que todas las instancias sean idénticas y predecibles.
- **Rollback Fácil:** Si surge un problema, es sencillo revertir al utilizar una versión anterior de la infraestructura.
- **Automatización:** Generalmente, se utiliza junto con herramientas de automatización y gestión de configuraciones como Terraform, Ansible o Docker.

Despliegues Blue-Green (Blue-Green Deployments)

El método de Despliegues Blue-Green implica mantener dos entornos de producción idénticos pero separados, denominados “Blue” y “Green”. Solo uno de estos entornos está activo y sirve el tráfico de usuarios en un momento dado.

1. **Ambiente Activo:** Supongamos que el entorno “Blue” está actualmente activo y sirviendo a los usuarios.
2. **Despliegue en el Nuevo Entorno:** Se despliega la nueva versión de la aplicación en el entorno “Green”, que está inactivo.
3. **Pruebas:** Se realizan pruebas exhaustivas en el entorno “Green” para asegurar que todo funcione correctamente.
4. **Cambio de Tráfico:** Una vez validado, el tráfico de usuarios se redirige del entorno “Blue” al “Green”.
5. **Entorno Anterior como Reserva:** El entorno “Blue” ahora puede mantenerse como una copia de seguridad o ser utilizado para el próximo despliegue.

Releases Canary (Canary Releases)

Las Releases Canary son una estrategia de despliegue progresivo donde una nueva versión de la aplicación se lanza inicialmente a un pequeño subconjunto de usuarios antes de desplegarla a toda la base de usuarios. El término “canario” hace referencia a la práctica minera de utilizar canarios para detectar gases tóxicos en minas.

1. **Despliegue Inicial:** Se lanza la nueva versión de la aplicación a un pequeño porcentaje de servidores o usuarios.
2. **Monitoreo:** Se monitorea de cerca el rendimiento y el comportamiento de la nueva versión.
3. **Evaluación:** Si no se detectan problemas significativos, se incrementa progresivamente el porcentaje de usuarios que utilizan la nueva versión.
4. **Despliegue Completo:** Una vez validada, la nueva versión se despliega a todos los usuarios.
5. **Rollback si es Necesario:** Si se detectan problemas, se puede revertir rápidamente al despliegue anterior sin afectar a la mayoría de los usuarios.

Diagrama de Arquitectura en la Nube

Computación

EC2

Instancias virtuales escalables para ejecutar aplicaciones en servidores dedicados.

Elastic Beanstalk

Servicio PaaS para desplegar aplicaciones web rápidamente con infraestructura gestionada.

AWS Lambda

Computación serverless para ejecutar código en respuesta a eventos sin gestionar servidores.

Fargate

Servicio para ejecutar contenedores en ECS o EKS sin gestionar servidores.

Compute Engine

Máquinas virtuales personalizables para ejecutar cargas de trabajo generales.

App Engine

Plataforma para desplegar aplicaciones web y backend en un entorno gestionado.

Cloud Functions

Servicio serverless para ejecutar funciones en respuesta a eventos.

GKE (Kubernetes Engine)

Servicio gestionado para ejecutar clústeres de Kubernetes.

Virtual Machines

Instancias de VMs para ejecutar sistemas operativos y aplicaciones personalizadas.

App Service

Servicio PaaS para alojar aplicaciones web y APIs sin gestionar servidores.

Azure Functions

Plataforma serverless para ejecutar código en demanda basado en eventos.

AKS (Kubernetes Service)

Servicio gestionado de Kubernetes para ejecutar y orquestar contenedores.

Almacenamiento

S3

Almacenamiento de objetos escalable y duradero para datos estructurados o no.

EBS

Almacenamiento en bloques para instancias EC2, ideal para bases de datos y aplicaciones.

Glacier

Almacenamiento en frío para datos que requieren archivado a largo plazo.

EFS

Sistema de archivos compartido y escalable para múltiples instancias.

Cloud Storage

Almacenamiento de objetos similar a S3 para datos escalables y duraderos.

Persistent Disks

Almacenamiento en bloques para VMs de Compute Engine.

Coldline Storage

Almacenamiento en frío optimizado para datos de acceso poco frecuente.

Filestore

Sistema de archivos compartido para instancias de Compute Engine.

Blob Storage

Almacenamiento de objetos optimizado para datos no estructurados y multimedia.

Azure Disk Storage

Almacenamiento en bloques de alto rendimiento para instancias de VM.

Azure Archive Storage

Almacenamiento de bajo costo para datos a largo plazo con acceso esporádico.

Azure Files

Sistema de archivos gestionado accesible desde máquinas virtuales y en la nube.

Bases de Datos

RDS

Servicio gestionado para bases de datos relacionales (MySQL, PostgreSQL, etc.).

DynamoDB

Base de datos NoSQL completamente gestionada para aplicaciones de alta demanda.

Aurora

Base de datos relacional compatible con MySQL y PostgreSQL optimizada para la nube.

Redshift

Almacenamiento de datos para análisis a escala de petabytes.

Cloud SQL

Servicio gestionado para bases de datos relacionales (MySQL, PostgreSQL, etc.).

Firestore

Base de datos NoSQL gestionada para datos estructurados y consultas en tiempo real.

Spanner

Base de datos relacional distribuida globalmente con transacciones ACID.

BigQuery

Almacenamiento de datos analíticos para consultas rápidas a gran escala.

Azure SQL Database

Base de datos relacional gestionada en la nube basada en Microsoft SQL Server.

Cosmos DB

Base de datos NoSQL global distribuida para baja latencia y alta disponibilidad.

SQL Managed Instance

Instancia gestionada de SQL Server con compatibilidad total para migraciones.

Synapse Analytics

Plataforma de análisis de datos integrando consultas SQL y big data.

Redes

VPC

Red privada virtual para alojar servicios y aplicaciones de manera aislada.

Elastic Load Balancing

Balanceo de carga para distribuir el tráfico entre instancias o contenedores.

Route 53

Servicio de DNS para gestionar nombres de dominio con alta disponibilidad.

Direct Connect

Conexión dedicada desde redes locales a AWS.

VPC

Red privada virtual para controlar redes en Google Cloud.

Load Balancing

Servicio gestionado para distribuir tráfico entre recursos.

Cloud DNS

Servicio de DNS escalable y gestionado para aplicaciones.

Interconnect

Conexión dedicada desde redes locales a Google Cloud.

Virtual Network

Redes virtuales para conectar recursos en la nube de manera segura.

Load Balancer

Balanceador de carga para distribuir tráfico entre servicios o máquinas virtuales.

Azure DNS

Servicio de DNS para alojar y resolver nombres de dominio.

ExpressRoute

Conexión privada dedicada entre redes locales y Azure.

Contenedores y Orquestación

Amazon ECS

Servicio de orquestación de contenedores que permite ejecutar y gestionar contenedores Docker en clusters.

Amazon EKS

Servicio gestionado para ejecutar Kubernetes, compatible con clústeres on-premises.

App Runner

Plataforma para desplegar y escalar automáticamente aplicaciones en contenedores.

Lightsail Containers

Solución sencilla para ejecutar contenedores en proyectos pequeños, con configuración mínima.

Google Kubernetes Engine (GKE)

Servicio completamente gestionado para ejecutar y orquestar contenedores usando Kubernetes.

Cloud Run

Plataforma serverless para ejecutar contenedores en respuesta a solicitudes HTTP.

Anthos

Plataforma híbrida y multicloud para gestionar clústeres de Kubernetes en diferentes entornos.

Cloud Build

Servicio de CI/CD para construir y desplegar imágenes de contenedores.

Azure Kubernetes Service (AKS)

Servicio completamente gestionado para ejecutar Kubernetes en la nube.

Azure Container Instances (ACI)

Servicio serverless para ejecutar contenedores sin la necesidad de gestionar servidores.

App Service

Permite ejecutar aplicaciones en contenedores, con soporte para Docker e imágenes personalizadas.

Azure Container Apps

Plataforma serverless para ejecutar microservicios y aplicaciones en contenedores.

Diagrama de Arquitectura en la Nube

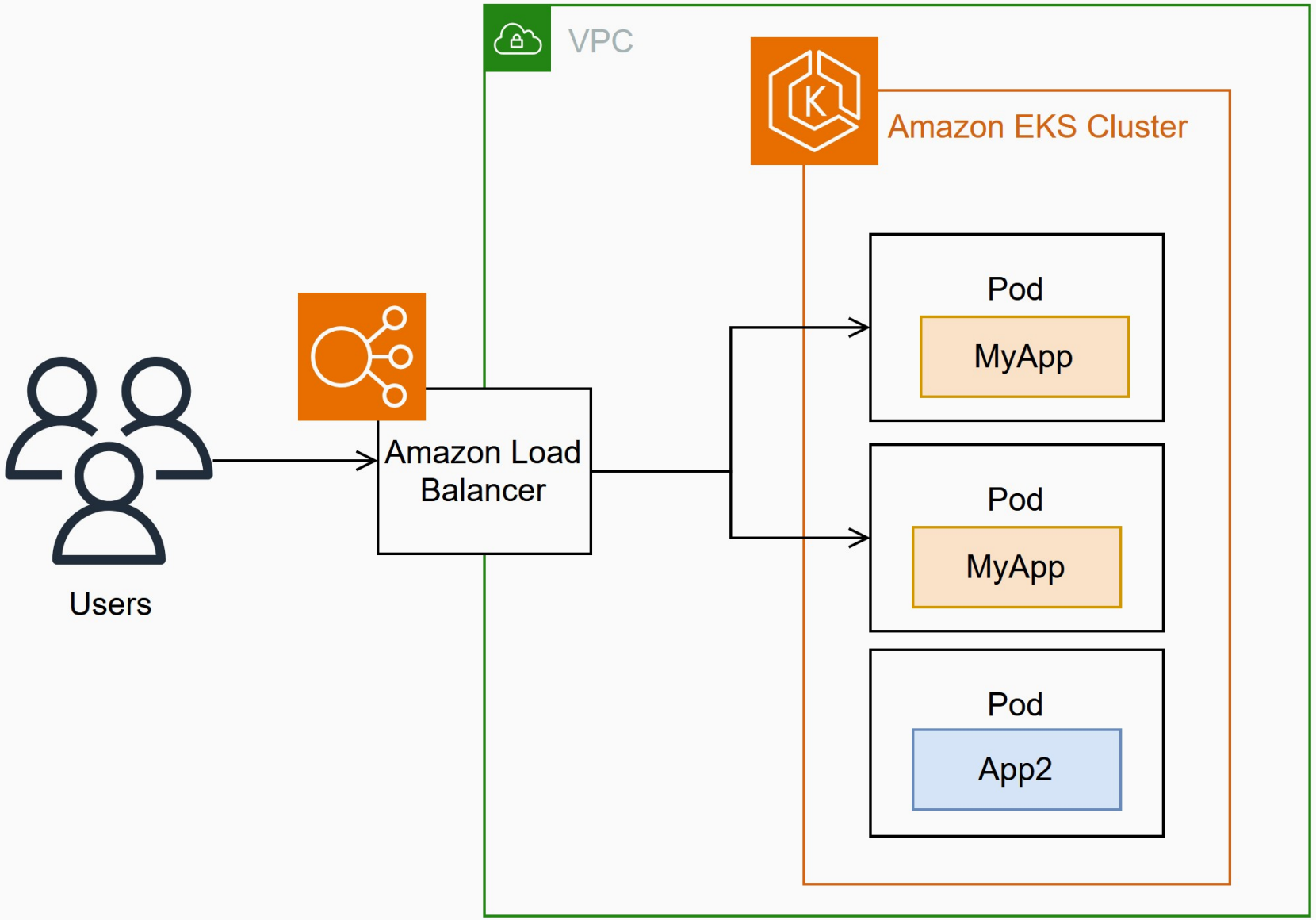


Diagrama de Arquitectura en la Nube

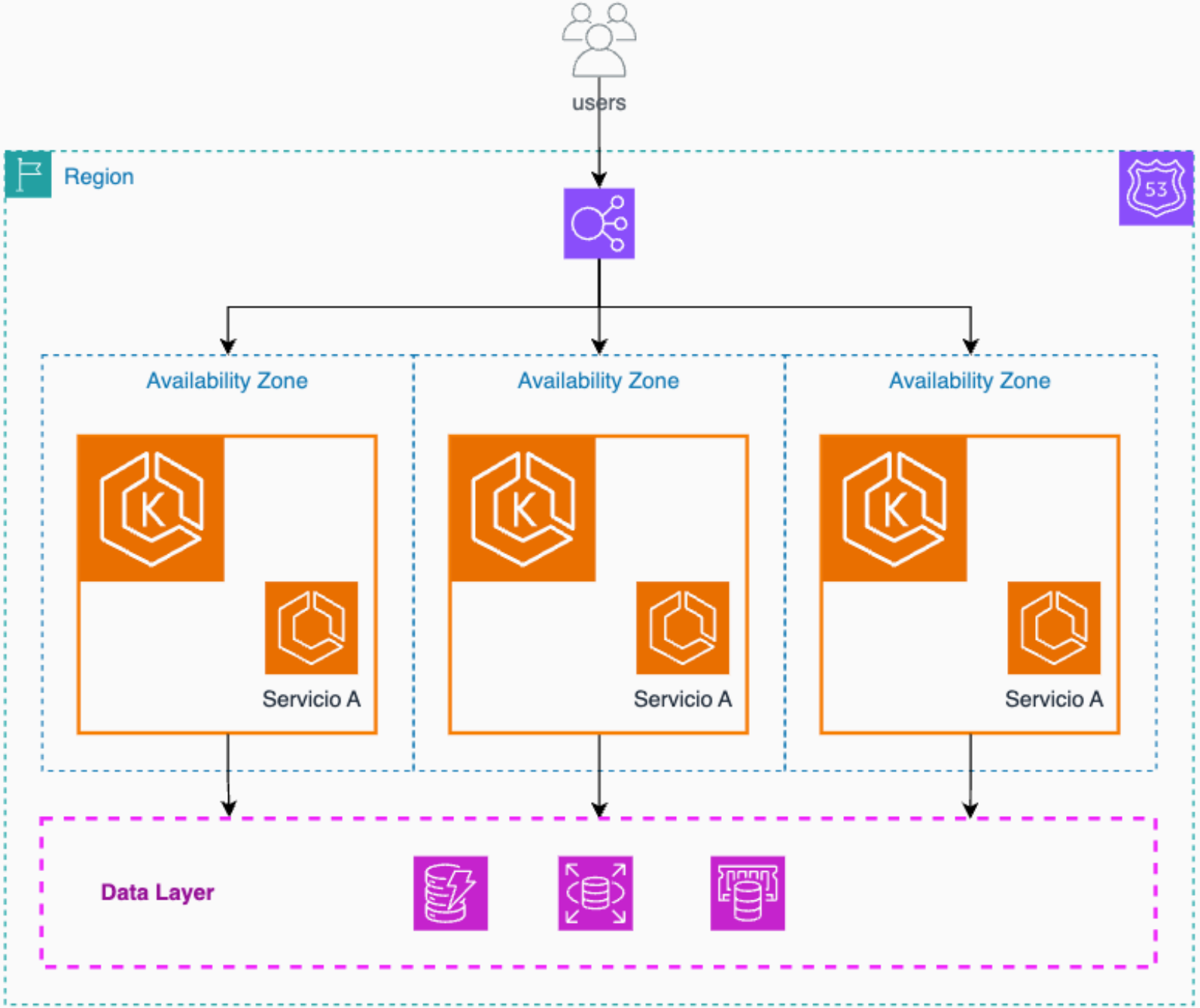
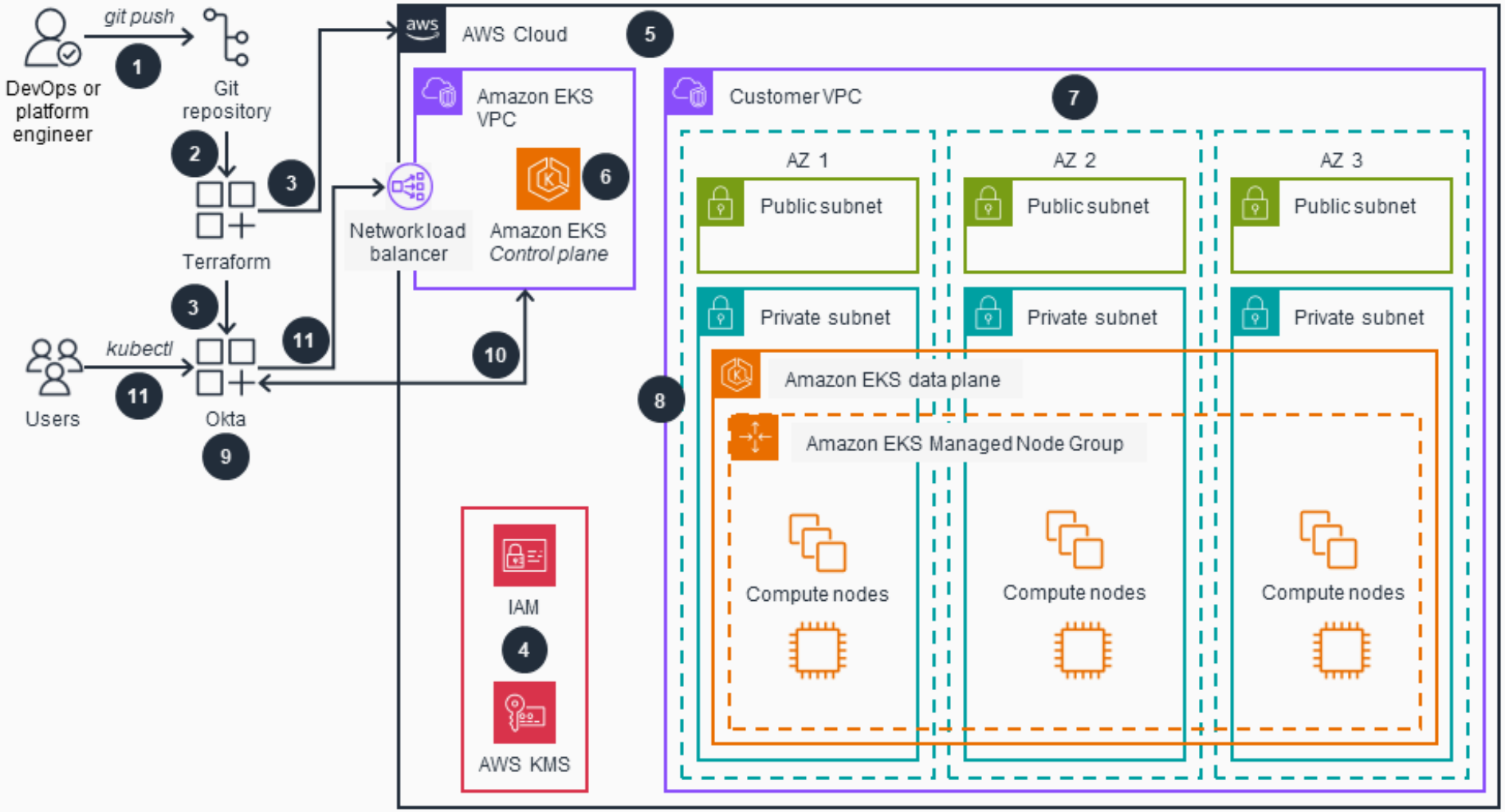
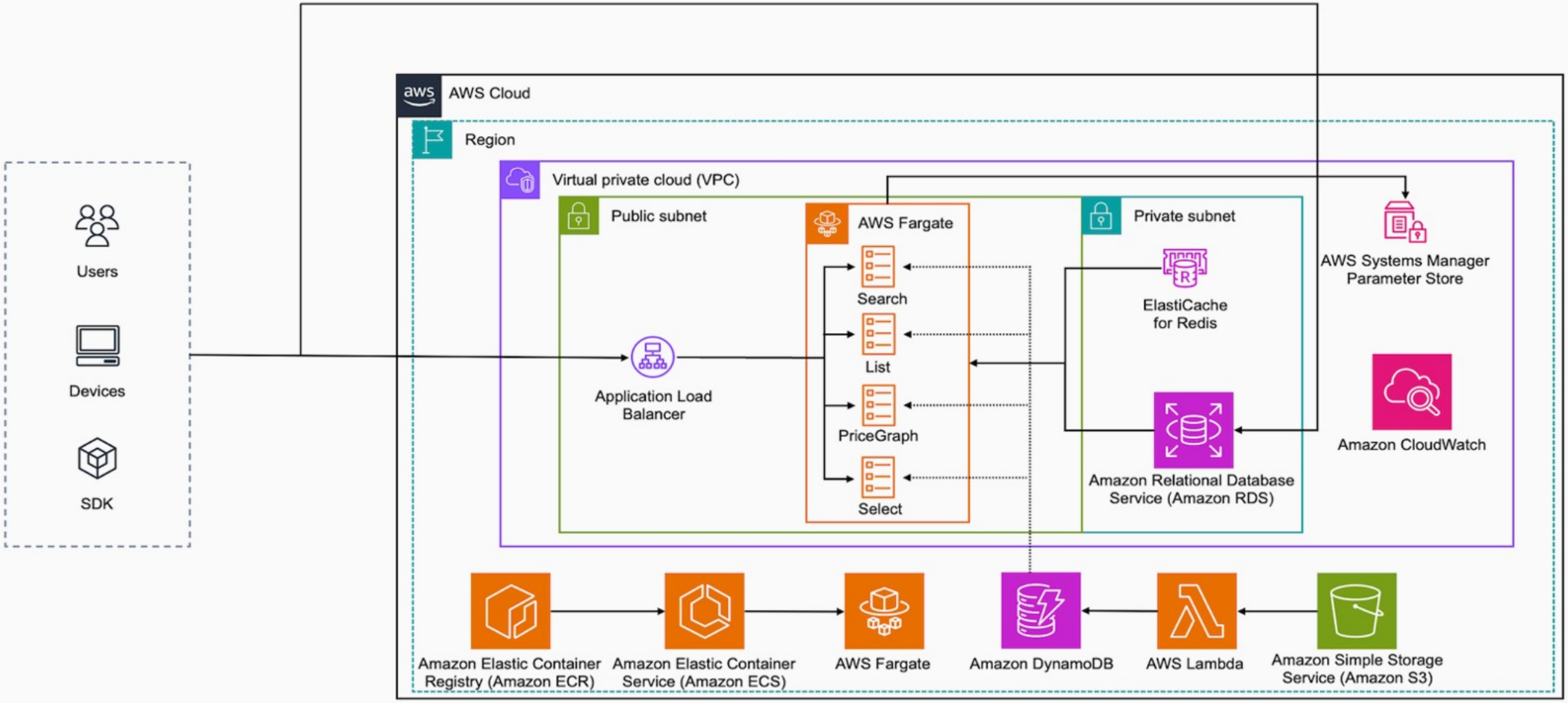


Diagrama de Arquitectura en la Nube




AWS Cloud
Region
Virtual private cloud (VPC)
Public subnet
AWS Fargate
Private subnet
Users
Devices
SDK
Application Load Balancer
Search
List
PriceGraph
Select
ElastiCache for Redis
Amazon Relational Database Service (Amazon RDS)
AWS Systems Manager Parameter Store
Amazon CloudWatch
Amazon Elastic Container Registry (Amazon ECR)
Amazon Elastic Container Service (Amazon ECS)
AWS Fargate
Amazon DynamoDB
AWS Lambda
Amazon Simple Storage Service (Amazon S3)